RESEARCH-ARTICLE

PG-Keys: Keys for Property Graphs

RENZO ANGLES, University of Talca, Talca, ML, Chile

ANGELA BONIFATI, Claude Bernard Lyon 1 University, Villeurbanne, Auvergne-Rhone-Alpes, France

STEFANIA DUMBRAVA, Polytechnic Institute of Paris, Palaiseau, Ile-de-France, France

GEORGE H L FLETCHER, Eindhoven University of Technology, Eindhoven, Noord-Brabant, Netherlands

KEITH W HARE

JAN HIDDERS, University of London, London, U.K.

View all

PG-Keys: Keys for Property Graphs

Renzo Angles
Universidad de Talca, IMFD Chile

Angela Bonifati
Lyon 1 Univ., Liris CNRS & INRIA

Stefania Dumbrava
ENSIIE & Inst. Polytechnique de Paris

George Fletcher
Eindhoven Univ. of Technology

Keith W. Hare
JCC Consulting Inc., Neo4j

Jan Hidders
Birkbeck, Univ. of London

Victor E. Lee
TigerGraph

Bei Li
Google LLC

Leonid Libkin
U. of Edinburgh, ENS-Paris/PSL, Neo4j

Wim Martens
University of Bayreuth

Filip Murlak
University of Warsaw

Josh Perryman
Interos Inc.

Ognjen Savković
Free Univ. of Bozen-Bolzano

Michael Schmidt
Amazon Web Services

Juan Sequeda
data.world

Sławek Staworko
U. Lille, INRIA LINKS, CRIStAL CNRS

Dominik Tomaszuk
Inst. of Comp. Sci., U. of Bialystok

ABSTRACT

We report on a community effort between industry and academia to shape the future of property graph constraints. The standardization for a property graph query language is currently underway through the ISO Graph Query Language (GQL) project. Our position is that this project should pay close attention to schemas and constraints, and should focus next on key constraints.

The main purposes of keys are enforcing data integrity and allowing the referencing and identifying of objects. Motivated by use cases from our industry partners, we argue that key constraints should be able to have different modes, which are combinations of basic restriction that require the key to be *exclusive*, *mandatory*, and *singleton*. Moreover, keys should be applicable to nodes, edges, and properties since these all can represent valid real-life entities. Our result is PG-Keys, a flexible and powerful framework for defining key constraints, which fulfills the above goals.

PG-Keys is a design by the Linked Data Benchmark Council's Property Graph Schema Working Group, consisting of members from industry, academia, and ISO GQL standards group, intending to bring the best of all worlds to property graph practitioners. PG-Keys aims to guide the evolution of the standardization efforts towards making systems more useful, powerful, and expressive.

CCS CONCEPTS

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

KEYWORDS

property graphs; key constraints

> *Brothers and sisters, I have none. But that man's*
> *father is my father's son. Who is that man?*
> –A Classic Riddle about Identity

1 INTRODUCTION

Graphs are a flexible and agile data model for representing complex network-structured data used in a wide range of application domains, including social networks [64], biological networks [52, 60], bioinformatics [58], cheminformatics [74], medical data [81], and knowledge management [41, 82]. In the context of enterprise data management, many current graph database systems (e.g., Amazon Neptune [4], Neo4j [70], TigerGraph [29]) support property graphs. A property graph is a multigraph where nodes and edges can have labels and properties (i.e., key-value pairs) [14].

The development of standards for property graphs is in process. In September 2019, ISO/IEC JTC1 approved a project to standardize a property graph database language GQL. The GQL project is assigned to ISO/IEC JTC1 SC32 WG3 Database Languages – the same committee responsible for developing and enhancing the Database Language SQL and of which four of this paper's authors are members. A standards effort for a new database model is a daunting task: the SQL standard was first approved in 1986 and continues to be enhanced and expanded today. In addition to standardizing the language for queries, the following aspects also need to be considered: extensions to the data model, schema language, constraints,

among others. It is unrealistic to expect that all of these features can be addressed by the ISO committee in a timely fashion. Therefore the initial focus of the ISO committee is to standardize a graph query language. This begs the question: what about standardization efforts for property graph schema and constraints?

Standards for property graph schema and constraints are critical to avoid interoperability risks. With the increased popularity of graph databases and the strong industry uptake, vendors are implementing their own versions of schema and constraints. By the time the ISO committee starts to address schema and constraints, the drift will be so large that it will be hard to reconcile approaches.

As a community of graph database industry practitioners and academics, we acknowledge the need for standardized property graph schema and constraints. Our work in the Linked Data Benchmark Council (LDBC) Property Graph Schema Working Group (PGSWG) is focused on providing community recommendations to the ISO committee. This paper presents the deliverable and recommendation of the working group around the notion of PG-KEYS, namely *keys for property graphs*. Intuitively, a key in a property graph database is used to establish and identify unique nodes, edges, and properties in the property graph.

Why Keys? Consider the following scenario to motivate the need for keys in a property graph. An e-commerce company is in its digital transformation process and in order to be competitive, it needs to offer customers new functionalities such as personalized recommendations based on their social network, new offers based on purchasing habits, etc. To accomplish this goal, the company wants to create an identity graph in order to achieve a 360 view of a customer and subsequently apply graph algorithms for next generation analytics. This identity graph is the result of integrating several data sources including the company's internal relational data (i.e., order management system, customer relationship management system) and acquired external data not necessarily in relational format (i.e., social media data and consumer behavior data in graph-oriented format). Notice that some data sources, such as the relational ones, might already have some constraints. Moreover, data can also be injected through ETL pipelines, where the constraints are enforced. However, even with cleaned data sources or input ETL pipelines, their integration and migration into the graph database might still bring global inconsistency. Thus, constraints should be persistent within the graph database, where the integrated and migrated data is stored, and should be seamlessly enforced on these data. Therefore, when it comes to integrate data from graph and non-graph data sources, the need of specifying constraints for the result of the integration is quite evident, as shown in the following use cases.

Example 1.1 (Identity in the graph). An identity graph providing a 360 view of customers must have a way to uniquely identify customer nodes in the graph. The order management system contains a unique identifier for a customer which is in the `customer_id` column of the `Customer` table. The following is an example (using a GQL-like syntax, quite similar with that of Cypher [36, 38]) of two `Customer` nodes in the property graph where `customerid` is the property that uniquely identifies a `Customer` node:

```
(:Customer {customerid:"C123", name:"Alice Smith"}),
(:Customer {customerid:"C456", name:"Jan" }).
```

Without an identity key, the resulting property graph could have `Customer` nodes that appear to be the same when in reality they are not. This would introduce data errors in the graph. With an identity key, the integrity of the data is maintained, meaning that there cannot be another `Customer` node with the same `customerid`.

The next source to incorporate is social media data containing data about Twitter users and the posts they have liked. The following is an example of a user node in the property graph, where username is a key that uniquely identifies a `TwitterUser` node:

```
(:TwitterUser {username:"asmith", firstname:"Alice",
               lastname:"Smith", email:"asmith@d.org"}).
```

In order to merge `Customer` and `TwitterUser` nodes that correspond to the same people, the company applies entity resolution techniques. The merged entity contains (deduplicated) data from both sources, for example:

```
(:Customer {customerid:"C123", name:"Alice Smith",
            twitter:"asmith", email:"asmith@d.org"}).
```

Note that not every `Customer` node would have corresponding social media data, so the following node remains perfectly valid:

```
(:Customer {customerid:"C456", name:"Jan" }).
```

Here, both `customerid` and `twitter` are exclusive, meaning that no two nodes should have identical values of either property. In addition, `customerid` is required for each node, while `twitter` is optional. Such semantics should ideally be encoded in the key constraints to prevent ill-formed data from entering the graph. If each `Customer` does not have a unique `customerid`, then the associated social media information for a single customer may be incorrectly merged into more than one customer:

```
(:Customer {customerid:"C456", name:"Alice Smith",
            twitter:"asmith", email:"asmith@d.org"}),
(:Customer {customerid:"C456", name:"Jan",
            twitter:"asmith", email:"asmith@d.org" }).
```

Example 1.2 (Integrity in the graph). An `Order` is placed by a `Customer`, as shown in the following graph pattern:

```
(:Order)-[:placedBy]->(:Customer).
```

Business users would want to ensure that (1) an `Order` must be associated with a `Customer` and be exclusive to that `Customer` and (2) a `Customer` can place zero or more `Orders`. These business rules can be ensured through key constraints. It is not uncommon that key constraints like this also imply such a participation constraint.

Without this key constraint for `Order`, the resulting property graph could have `Order` nodes without an associated `Customer`, thus introducing data errors in the graph. For example, if in the process of replicating the order management system to a property graph, the `Order` table has a `customer_id` column, which is a foreign key referencing the `Customer` table, and the `customer_id` column is nullable, this can lead to problems. Perhaps that value is allowed to be changed later, or maybe it is a bug in the order management system. If a row in the `Order` table has a NULL value in the `customer_id` column, then the resulting property graph can have an `Order` node without a corresponding `Customer`. This would heavily affect analytics and recommendations and subsequently create the need to invest more money and effort in data cleaning.

With the key constraints, these problems would be avoided. First, the keys enforce the data integrity of the property graph. Second, data quality issues can be identified proactively thus avoiding costly data cleaning expenses later on. Therefore, business rules can be

modeled as key constraints, thereby maintaining property graphs consistently and preventing data quality issues.

The above use cases showing the utility of PG-Keys in data integration and data migration pipelines are recurrent in graph database applications, as witnessed by the industry members of the PGSWG (namely Amazon, data.world, Google, Interos, Neo4j, and TigerGraph). Furthermore, property graph databases such as Neo4j, Tigergraph, etc., are transactional, therefore they can be used as a database of record. For this reason, keys are crucial to reference and identify a node, edge, or property in a graph, avoid duplicate nodes and edges, provide a base for describing how one entity connects to another, constrain the structure of the database, and enforce identity and integrity of the nodes, edges, and properties in a graph, among others.

From an academic viewpoint, keys are interesting to study because they are the most basic and most-used type of database constraint, and as such play a fundamental role in reasoning over data and queries for the sake of correctness and performance. For example, the existence of keys influences the choice of data structures and algorithms for indexing, and they can be crucial for determining if a certain query always has a unique and meaningful result. So an effective and well-understood formalism for describing keys is crucial for developing a body of knowledge that can help with building correct and more efficient databases.

Property graph keys today. Given the popularity of property graphs and the rise of numerous database vendors, one would believe that incorporating keys in property graphs would be a foregone conclusion. Unfortunately, we are at a stage where there is already a significant drift between database vendors. From an academic perspective, there has been broad research of keys in a variety of data models, including graphs; however, the results are disconnected from the needs of industry.

Industry. The online documentation of thirteen property graph database systems (AgensGraph [3], Amazon Neptune [4], Azure Cosmos [59], DataStax [27], JanusGraph [45], Memgraph [57], Neo4j [76], Oracle Spatial and Property Graph [62], RedisGraph [47], Sparksee [75], TigerGraph [77], TinkerPop [9] and Titan [78]) reveal the following. Some systems (DataStax, Oracle Spatial and Property Graph, and TigerGraph) offer primary keys for nodes, which combine three constraints on the property values: unique, mandatory, and single-valued. Other systems take a more granular approach. AgensGraph, Memgraph, Microsoft Azure Cosmos, Neo4j, and Sparksee support a uniqueness constraint on nodes: the same property value may not appear in more than one node of a given label or type. AgensGraph, Memgraph, and Neo4j also support a mandatory constraint: every node having a given label or type must have a value for the given property. Both uniqueness and mandatory are supported in Oracle relational database, which are then inherited in the derived Spatial and Property Graph. Some systems offer some of these constraints for edges, but no clear pattern emerges. In general, every edge has an implied constraint, namely that it connects one source node and one destination node. For TigerGraph, this is extended to say that these two nodes form the primary key of an edge. This means there may be only one edge of a given type between a given pair of nodes. Other systems do not have this built-in constraint, embracing a multigraph model. To our knowledge, none of these systems offer

user-specified edge cardinality. It is clear that there is no uniform approach taken for keys in property graphs. This lack of uniformity underlines the importance of a standardization effort motivated by existing use cases and based on solid theoretical foundations.

Academia. The notion of "key" exists in most database systems, although its definition and meaning depend on the underlying database model. For example, a key in the relational model is a set of attributes whose values are used to identify tuples inside a table [2]; in object-oriented data models, object identity is achieved by equipping each database object with a unique identifier [67] or by using an arbitrary query [65]; in semi-structured models and XML, a key can be specified in terms of path expressions [18]; in RDF, every resource is either identified by an Internationalized Resource Identifier (IRI) [48] or by using a term described by a vocabulary [49]. Section 5 gives a closer review of these notions.

In the context of graph databases, we can also find different approaches to key constraints in the research literature [8, 14]. A prominent proposal for graph keys [33] relies on a graph model that differs from the property graph model in that property values are modeled as data value objects (i.e., special data nodes) and edges do not have identity or property values. Keys there are a kind of uniqueness constraint defined in terms of graph patterns (to specify topological constraints and value bindings), and are interpreted based on graph pattern matching. The focus of the paper is on the entity-matching problem rather than on producing a widely applicable recommendation for the design of keys, rooted in industrial property graph use-cases. Specifically, the authors analyze the general complexity of the problem and evaluate two specific algorithms for entity matching. The work has been extended to graph functional dependencies [34], though for a model that is less general than property graphs, as values are only allowed at the nodes. The extension focuses on the satisfiability, implication and validation problems for functional dependencies rather than on a recommendation proposal for property graph databases.

A recent formalism [51] allows the definition of property-based key constraints solely on nodes and discusses its possible implementation on top of Neo4j using the Cypher language; that is, for a label and a set of properties, all the properties must exist in all the nodes with that label, and each combination of the values for these properties is unique for each node. It is also observed that such keys can be extended to nodes that carry multiple labels. The focus of this work is on the implication problem and its axiomatic characterization. Another extension of Neo4j constraints [68] studies new notions such as "node property uniqueness" to make reference to a set of properties whose values must be unique for a given set of nodes, and "mandatory properties" for nodes and edges.

Although various aspects of key constraints for graph database models have been touched upon in the past, a full-fledged formalism as the one presented in this paper is missing. Moreover, it is clear that there is no consensus among the vendors. The existence of this working group is evidence that a consensus is needed.

Contributions. Motivated by the current situation in property graph data management systems, and following the success of G-CORE [6], the Property Graph Schema Working Group was formed in 2019. This paper documents the consensus based on intensive and constructive discussions held over 18 months between the industry

and academic members of the group. Our contributions are: (a) an analysis of the requirements for property graph keys; (b) a proposal for a modular, flexible, and expressive formalism called PG-Keys, that defines a syntax and a semantics for specifying keys, satisfying all design requirements; (c) a comparison of this formalism with keys in existing database models and data models.

Our contributions impact the following audiences: (1) industry practitioners building graph databases, who can use our framework as a guideline to incorporate keys in their systems, (2) graph database standards committee members, who can build upon our recommendations for upcoming standardization features, and (3) academics, who are given a concrete model of keys for property graphs, which they can use as a basis for further research.

2 DESIGN REQUIREMENTS FOR KEYS

In this section we elaborate on the design requirements for a suitable notion of key for property graphs. We begin by discussing the relevant functions that keys play in databases.

2.1 Purposes of Keys

A common reason for using keys is to **constrain** the database contents and prevent data patterns that are nonsensical, contradictory, or unnatural. For instance, a key can be used to prevent a database from storing two copies of information for individuals using the same SSN (Social Security Number). Also, in relational tables that represent relationships between entities, keys are used to express *participation constraints* that restrict the relationships to many-to-many, one-to-many, and one-to-one kind, cf. Example 1.2.

Another purpose of keys is to allow one to **reference** database objects. For example, foreign keys in relational databases allow one record to reference another, by citing its primary key, a common mechanism for representing relationships between entities. In property graphs, relationships are represented with edges rather than foreign keys, and consequently, keys are not needed for intra-database referencing. However, a reference mechanism is still required by external applications that access the database.

A special, but distinct, case of referencing is when keys are used to **identify** real-life objects represented by database objects, and vice versa. To this end, keys specify the identifying information for each object. This use case is particularly relevant in various entity resolution problems [23, 31], where it is essential that the identities of objects can be compared through their identifying information (cf. Example 1.1). Also, this use of keys is crucial in conceptual data models and object-oriented database models [42, 43], which typically introduce an abstract object identifier and a link to real-life objects needs to be established.

All of the above uses of keys are relevant for property graphs and motivate the following requirement.

R0 **Coverage.** The proposed formalism must address the need to constrain the database and to reference and identify objects.

2.2 Key Types

To properly address the various purposes of keys, we elaborate on a repertoire of key types of varying power, but first we need to outline the basic anatomy of keys and how they work.
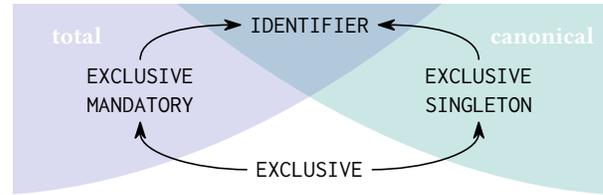

Figure 1: Hierarchy for PG-Keys.

A key has a *scope*, which is the set of objects to which it applies, and a *descriptor*, which specifies, for an object in the key scope, how to obtain a *key value*. For instance, in relational databases, the scope of a key is a table and the descriptor consists of a set of (key) attributes: for an object represented by a single row of a table, the key value consists of the values of key attributes in that row.

For object identification purposes, the key values need to be:

EXCLUSIVE, no two objects in the key scope can share a key value. This ensures unambiguity of the reference given by the key.

MANDATORY, every object in the key scope must have a key value. This ensures that every object in the key scope can be referenced using the key, which provides a *total* reference scheme.

SINGLETON, every object in the key scope must have at most one key value. This ensures that the key value is *canonical*, and in particular, for any two objects in the scope that have key values, the objects are identical if and only if their key values are the same. So, the key value of an object is equivalent to its identity.

The conditions above are in fact fulfilled by virtually all existing notions of keys, including keys in relational databases [2], keys in ER diagrams [22], and keys in XML Schema [20, 53].

For the purposes of referencing objects and imposing constraints, we additionally consider variants that drop either MANDATORY or SINGLETON or both. Indeed, such variants are commonly employed in existing database models and data modeling frameworks. For instance, XML Schema proposes the <unique> identity constraint that drops MANDATORY. Similarly, SQL provides a UNIQUE constraint for relational databases (where SINGLETON is implied by 1NF).

For the presented purposes, we identify four natural key types, which we illustrate with an example of a system that manages information about a set of users:

- EXCLUSIVE MANDATORY SINGLETON, or IDENTIFIER for short, e.g., *login*: every user is required to have precisely one, and no two users can have the same login;
- EXCLUSIVE MANDATORY, e.g., *email*: every user must have at least one email and no two users can use the same email;
- EXCLUSIVE SINGLETON, e.g., *preferred email*: every user may have at most one preferred email for contacting them but again no two users can have the same preferred email;
- EXCLUSIVE, e.g., *alias*: every user may have an arbitrary number of aliases but no two users can have a common alias.

These four key types form a natural hierarchy, presented in Figure 1: arrows lead from weaker to stronger types of keys.

2.3 Defining Scope and Descriptor

We next identify design requirements that focus on defining the scope and the descriptor of keys. To illustrate them, we use a small example of a property graph, presented in Figure 2, representing a

Social Network (SN) graph database inspired by the LDBC Social Network Benchmark [5, 32]. Recall that a property graph is a directed labeled multigraph whose nodes and edges have (possibly multiple) labels and properties [7, 8, 14].

For a majority of keys, the scope and the descriptor are defined by simple means of inspecting labels and property values only. For instance, countries, represented by nodes with label Country, are identifiable by their name, stored as the property name.

However, the information relevant to establishing the kind of a node and its key value may be located outside of the node: accessing it may require navigating through the graph. For instance, suppose that we wish to express a key that asserts that forum moderators can be distinguished through their first and last names alone. The scope of such a key consists of Person nodes with an incoming hasModerator edge. Similarly, consider a key stating that a city can be identified with the combination of its name and its country: the key descriptor needs to access a Country node reachable from the City node with an outgoing isPartOf edge. Hence, the following.

R1 **Key Scope.** The proposed key formalism must support a rich language that allows specification of relevant elements of the property graph that represent real-world objects. In particular, it must allow the selection of nodes, edges, and their properties. This language cannot assume that a schema is present.

R2 **Key Descriptor.** Additionally, the proposed key formalism must support an equally rich language to locate the graph elements constituting the key value of an object in scope.

2.4 Object Identity

In R1, we state the need to consider nodes, edges, and properties as the objects which one may wish to identify with a key. The justification for nodes is straightforward, since they are typically used to represent real-world objects. Edges represent relationships, which may capture events and facts. For instance, in the SN graph database, the studyAt edges represent the fact that an individual attends university and, as such, we may wish to identify them. Finally, it is also conceivable that a real-world object is not represented directly in a graph database with a dedicated node or edge, but rather by a property value. For instance, a mobile phone may be represented by its IMEI number, stored as an attribute of the node representing its owner. We point out, however, that property values are literal values and, hence, do not have an identity of an abstract data object, such as nodes and edges. Consequently, the treatment of property values needs to adequately address this difference.

In the context of the relational model, keys are modeled as equality generating dependencies [2], where the generated equalities are between domain values. However, they can also be understood as a way of determining if two records represent the same real-world object. Namely, any two records that agree on key attributes represent the same real-world object and, therefore, should be equal. Keys for property graphs can also be understood in this way as a mechanism for determining the identity of the objects by nodes, edges, and property values. This leads to the following requirements.

R3 **Node Identity.** Our formalism must allow determining the identity of nodes in a graph database.

R4 **Edge Identity.** Our formalism must allow determining the identity of edges in a graph database.

R5 **Property Value Identity.** Our formalism must also allow identification of property values of both nodes and edges. In particular, the formalism must allow the determination that two properties must or must not have the same value.

Note the distinction between object identity and the ability of users to observe and compare object identifiers (i.e., the concrete object ID values used internally by a system implementation). Towards maximal flexibility for system designers and implementors, we do not require access to observable object identifiers in our formalism.

2.5 Pragmatic Concerns

Finally, we list requirements of a pragmatic nature. They are concerned with ease of use and feasibility of implementation.

R6 **Usability.** The keys defined by the formalism must be understandable and intuitive for the intended users. Preferably the formalism should be declarative.

R7 **Validation.** It should be relatively straightforward to validate a key, i.e., check whether it holds in a given property graph. Its complexity should be comparable to the complexity of executing a query in the available querying apparatus.

3 QUERYING PROPERTY GRAPHS

In this section, we discuss languages that can be used to specify the scope and descriptor of key constraints for property graphs. To this end, we first treat property graphs themselves. A property graph is a directed labelled multigraph with the special characteristic that each node or edge maintains a (possibly empty) set of *properties*, where a property is a name-value pair. From a data modeling point of view, a node represents an entity, an edge represents a relationship between entities, a label represents a classification or type, and a property represents an attribute of an entity or relationship.

The general structure of a property graph can be restricted to satisfy specific requirements. In this paper we will assume the following restrictions: each node/edge has an exclusive object identifier (oid); each node/edge has zero or more labels; each node/edge has zero or more properties; the value of a property must be either a simple value (e.g., a number, a string, a date) or a complex value (e.g., a tuple, a set, a JSON structure); and two properties (inside a single node/edge) cannot have the same name.

We now give a formal definition of property graphs. Assume that $\mathcal{L}$ is a countably infinite set, containing *labels* and *property names*, and $\mathcal{V}$ is a countably infinite set of *property values*.

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 $N \cap E = \emptyset$; $\rho : E \to (N \times N)$ is a total function mapping edges to ordered pairs of nodes; $\lambda : (N \cup E) \to 2^{\mathcal{L}}$ is a total function mapping nodes and edges to sets of labels (including the empty set); $\pi : (N \cup E) \times \mathcal{L} \nrightarrow \mathcal{V}$ is a partial function mapping nodes / edges and property names to property values.

Example 3.2. Consider the property graph in Figure 2. We have $N = \{ct_2, cn_2, \ldots, m_2, m_1\}$; $E = \{po_2, r_1, \ldots, s_2\}$; $\rho(po_2) = (ct_2, cn_2)$, $\ldots$, $\rho(r_1) = (m_2, m_1)$; $\lambda(ct_2) = \{\text{City}\}$, $\ldots$, $\lambda(m_1) = \{\text{Message, Post}\}$, $\lambda(po_2) = \{\text{isPartOf}\}$, $\ldots$, $\lambda(s_2) = \{\text{studyAt}\}$; and $\pi(ct_2, \text{name}) = \text{Wassenaar}, \ldots, \pi(s_2, \text{classYear}) = 2021$.
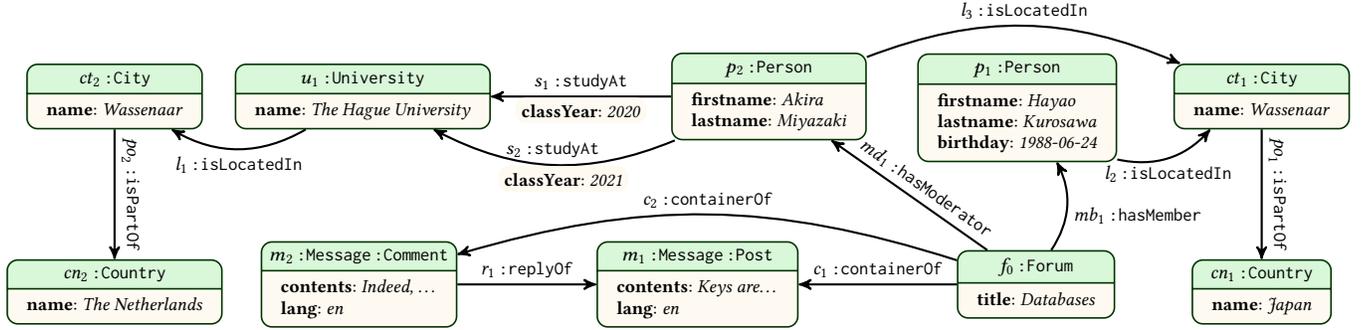
Figure 2: An example of a property graph: a Social Network (SN) graph database. For ease of reference, we associate with every node and edge an identifier and employ consistent typographic conventions. For instance, take the node u_1 representing the Hague University: u_1 is its identifier, University is its only label, **name** is a property name, and *The Hague University* is its value.

For defining the scope and descriptor of key constraints, we assume that we can use a language that allows us to map property graphs G to tables T, where a table T is a set of bindings that map variables to values. Concretely, we assume that we can write statements such as:

$$q(\bar{x}) \ ,$$

where $\bar{x}$ is a tuple of variables that bind to *nodes*, *edges*, and *property values*. We will use these statements to describe the *scope* and *descriptor* of keys. Informally, such a statement could be the query

"Return all bindings to (x, y, z) such that x is a person, y the city that x is located in, and z is the name of y."

On the data in Figure 2, this query would return the table

x	y	z
p_1	ct_1	Wassenaar
p_2	ct_1	Wassenaar

where the first and second row express that (x, y, z) can be bound to $(p_1, ct_1, \text{Wassenaar})$ and $(p_2, ct_1, \text{Wassenaar})$, respectively.

Notice that our convention to let variables bind to nodes, edges, and property values implies that (a) queries cannot output paths and (b) queries can only output *entire* property values, treating them as atomic. We will consider extensions of our formalism that relax these conditions in Sections 6.2 and 6.3.

The language for specifying $q(\bar{x})$ will be a *parameter* of PG-Keys. This means that different database systems can use different languages for specifying $q(\bar{x})$ and still fully conform to PG-Keys. We recommend using languages with a good expressiveness/complexity balance, which will allow efficient implementations of key validation, while providing sufficient expressive power.

In order to be able to present examples in the paper, however, we will specify $q(\bar{x})$ as queries in a language where the patterns are expressed in a GQL-like syntax, similar to that of Cypher [36], a popular graph query language. In this syntax the above query would be written as

```
x, y, y.name WITHIN
    (x:Person)-[:isLocatedIn]->(y:City).
```

Here, the part preceding keyword WITHIN specifies the output of the query, whereas the part following WITHIN specifies the pattern to be matched in the property graph. Notice that our syntax does not require giving an explicit name to every variable. For instance,

we just use y.name to refer to "the name of y", which we called z before. We will use this convention throughout the paper.

If there is exactly one output variable and this is the only variable in the pattern, as for example in x WITHIN (x:Person), then we allow the query to be specified by just the pattern (x:Person).

We will also assume that all variables in the query, including the implicit ones, must be bound to existing objects in the property graph. In Section 6.1, we discuss what happens when a reference to an undefined property is allowed to occur. In that case, following the practice of existing graph query languages, a null (or more precisely, a non-applicable null) is returned. Existing query languages tend to follow SQL's three-valued approach to handling nulls, though in Section 6.1 we suggest another approach that fits in better with the semantics of undefined properties: namely, returning *false* for results of comparisons using such properties.

4 A GUIDED TOUR OF PG-KEYS

In this section, we define PG-Keys formally and demonstrate how they satisfy the design requirements identified in Section 2.

We begin with a basic example illustrating the general shape of PG-Keys. Suppose that cities can be identified by their name and the country they are in (if this information is known). More precisely, this means that the combination of the name property of a city node, with the country node to which it has an isPartOf edge, identifies the city node. The corresponding PG-Key

```
FOR (x:City) EXCLUSIVE x.name, z WITHIN
    (x)-[:isPartOf]->(z:Country)
```

involves two queries. The query (x:City) specifies the *scope* of the PG-Key, which is the set of all possible *targets*; here, city nodes. The query x.name, z WITHIN (x)-[:isPartOf]->(z:Country) is the *descriptor* that selects a *key value* for each target; here, the city's name and the country it is part of. The keyword EXCLUSIVE indicates that the PG-Key asserts that the key value is exclusive to each target.

4.1 PG-Keys Formally

A *PG-Key* is an expression of the form

```
FOR p(x)
EXCLUSIVE [ MANDATORY | SINGLETON ] | IDENTIFIER  q(x,ȳ) ,
```

where $\bar{y} = (y_1, y_2, \ldots, y_n)$ for some positive integer n, and $p(x)$ and $q(x, \bar{y})$ are queries, called the *scope* and the *descriptor*, respectively.

Note that the keyword WITHIN appearing in the basic example belongs to our syntax for queries.

The keywords EXCLUSIVE, MANDATORY, and SINGLETON indicate which assertions the PG-KEY makes:

EXCLUSIVE — no two targets can have the same key value;
MANDATORY — for each target there is at least one key value;
SINGLETON — for each target there is at most one key value.

More precisely, the assertions can be formulated as follows:

(K1) for all o_1 and o_2 such that $p(o_1)$ and $p(o_2)$, for all $\bar{r}$ such that $q(o_1, \bar{r})$ and $q(o_2, \bar{r})$, it holds that $o_1 = o_2$;

(K2) for all o such that $p(o)$, there exists $\bar{r}$ such that $q(o, \bar{r})$;

(K3) for all o such that $p(o)$, for all $\bar{r}_1$ and $\bar{r}_2$ such that $q(o, \bar{r}_1)$ and $q(o, \bar{r}_2)$, it holds that $\bar{r}_1 = \bar{r}_2$.

A graph G satisfies: an EXCLUSIVE constraint if condition **(K1)** holds, an EXCLUSIVE MANDATORY constraint if **(K1)** and **(K2)** hold, an EXCLUSIVE SINGLETON constraint if **(K1)** and **(K3)** hold, and an IDENTIFIER constraint if **(K1)**, **(K2)**, and **(K3)** hold. That is, IDENTIFIER is a shorthand for EXCLUSIVE MANDATORY SINGLETON. PG-KEYS clearly satisfies design requirements R1–R5 of Section 2: there is full support for specifying the scope (R1) and descriptor of a key (R2); and, key constraints can be defined for nodes (R3), edges (R4), and property values (R5).

In the following Sections 4.2–4.4, we give a guided tour highlighting how design requirement R6 (Usability) is also satisfied, through the intuitive declarative way in which PG-KEYS are specified. We further address design requirement R0 (Full coverage): PG-KEYS allows us to constrain the graph database, reference objects within the database, and identify objects in the database. Indeed, our presentation follows the modular structure of the PG-KEYS formalism, showcasing the support provided for finely controlling the scope and descriptor (which can be complex queries over object properties and graph topology) and for the four key types identified in Section 2. Last but not least, the discussion of validation of PG-KEYS in Section 4.5 addresses design requirement R7 (Validation).

4.2 Keys on Nodes

Keys Defined Using Properties. Suppose that we are in the process of building our SN graph, and not all country nodes have name property values yet. However, the name property should be unique for each country, for countries that already have a name. More precisely, if it exists, the name property of a country node should identify the country node. This is a uniqueness (or exclusivity) constraint, allowing us to reference countries:

```
FOR (x:Country) EXCLUSIVE x.name .
```

In other words, given any two nodes n_1 and n_2 labeled Country, if they have the same value for the property name, then it must be the case that $n_1 = n_2$.

As the data becomes more complete, suppose that we further require that each country must have a name. In this case, reference constraints and identification constraints are equivalent, because our data model does not include multi-valued properties:

```
FOR (x:Country) EXCLUSIVE MANDATORY x.name ,
FOR (x:Country) IDENTIFIER x.name .
```

Keys Defined Using Properties and Topology. To further illustrate the distinction between EXCLUSIVE MANDATORY and IDENTIFIER,

we return to the example with which we opened this section, where cities are identified by their name and the country they are part of (if this information is known). This is an example of a uniqueness constraint allowing us to reference cities, i.e., given any two nodes n_1 and n_2 labeled City, if they have the same value for the property name and both have an isPartOf edge to a common node n_3 labeled Country, then it must be the case that $n_1 = n_2$. If we further require that cities must have names and must be part of a country, then it would be natural to specify the constraint

```
FOR (x:City) EXCLUSIVE MANDATORY x.name, z WITHIN
    (x)-[:isPartOf]->(z:Country)
```

allowing to reference cities and impose the required constraints on the graph topology. If we further require that cities are part of exactly one country, we would specify the identification constraint:

```
FOR (x:City) IDENTIFIER x.name, z WITHIN
    (x)-[:isPartOf]->(z:Country) .
```

Keys Defined Using Topology. As an example of a constraint defined purely in terms of graph topology, consider forums which can be identified by the posts that they contain, i.e., knowing a post, the forum that contains it is uniquely identified. This is a uniqueness constraint, allowing us to reference forums:

```
FOR (x:Forum) EXCLUSIVE z WITHIN
    (x)-[:containerOf]->(z:Post) .
```

In other words, given any two nodes n_1 and n_2 labeled Forum, if they both have a containerOf edge to the same Post, then it must be the case that $n_1 = n_2$. If we further require that forums must have posts, we specify an EXCLUSIVE MANDATORY constraint on the database. Further, it is not expected that forums must each have exactly one post (since these would be rather lonely forums), hence it doesn't make sense that posts are identifiers for forums. We can express that each post is contained in exactly one forum as

```
FOR (z:Post) MANDATORY SINGLETON x WITHIN
    (x:Forum)-[:containerOf]->(z) ,
```

but this is a participation constraint rather than a key constraint, and is not part of PG-KEYS.

Keys With Complex Scope. So far, we have defined constraints on nodes based on a fairly simple scope, namely, by only considering the label of the node. As a final example illustrating the need for more complex scope, consider the constraint that a forum which has members (1) must have a moderator and (2) is identified by the moderator. In PG-KEYS, we have

```
FOR x WITHIN (x:Forum)-[:hasMember]->(:Person)
    IDENTIFIER p WITHIN (x)-[:hasModerator]->(p:Person) .
```

In other words, a forum with members must have exactly one moderator and, furthermore, given any two such forum nodes n_1 and n_2, if they both have a hasModerator edge to the same Person, then it must be the case that $n_1 = n_2$.

4.3 Keys on Edges

Keys Defined Using Topology. For our first edge key constraint, consider that there is only one isPartOf edge from a given country to a given continent, i.e., the identity of an isPartOf edge from a country to a continent is determined by the country and the continent. More formally, this actually means that there is at most one isPartOf edge from a given country to a given continent, which is a uniqueness constraint:

```
FOR y WITHIN (:Country)-[y:isPartOf]->(:Continent)
EXCLUSIVE x, z WITHIN (x:Country)-[y]->(z:Continent).
```

In other words, given any two edges e_1 and e_2 labeled `isPartOf`, if they have the same source node n_s labeled `Country` and the same target node n_t labeled `Continent`, then it must hold that $e_1 = e_2$.

Keys Defined Using Properties and Topology. Suppose that people can study at the same university in different years, but for a given year, the `studyAt` edge between a person and a university is unique. Rephrased, this means that, in a given year, the information that a person studies at a given university is stored only once. More precisely, if you have a `studyAt` edge from a person to a university with the property `classYear`, then this edge is identified by the person, the university, and the value of `classYear`. This is a uniqueness constraint, which we can express in PG-Keys as

```
FOR y WITHIN (:Person)-[y:studyAt]->(:University)
EXCLUSIVE x, y.classYear, z WITHIN
  (x:Person)-[y]->(z:University).
```

In other words, given any two edges e_1 and e_2 labeled `studyAt` with property `classYear`, if they have the same source node n_s labeled `Person`, the same target node n_t labeled `University`, and have the same value for the property `classYear`, then it must be the case that $e_1 = e_2$.

As another example, suppose that `studyAt` edges *must* have a `classYear` property (and always be from person nodes to university nodes). As our properties are single-valued and edges have a single source and a single target, we have an identification constraint:

```
FOR y WITHIN ()-[y:studyAt]->()
IDENTIFIER x, y.classYear, z WITHIN
  (x:Person)-[y]->(z:University).
```

Note that this is not the same as:

```
FOR y WITHIN (:Person)-[y:studyAt]->(:University)
IDENTIFIER x, y.classYear, z WITHIN (x)-[y]->(z),
```

which has the scope limited to those `studyAt` edges that are from `Person` to `University`.

As a final example, suppose that our edge constraint only holds for study years after 1970. We can express this as

```
FOR y WITHIN ()-[y:studyAt]->() WHERE y.classYear > 1970
IDENTIFIER x, y.classYear, z WITHIN
  (x:Person)-[y]->(z:University).
```

4.4 Keys on Properties

We close our tour of the functionality of PG-Keys with an illustration of a constraint on properties. Consider that study semesters belong to a particular year, e.g., the first semester of 2019. That is, the `classYear` property of a `studyAt` edge is identified by the `semester` property of the edge (if it is known):

```
FOR y.classYear WITHIN ()-[y:studyAt]->()
EXCLUSIVE y.semester.
```

In other words, for any `classYear` property values v_1 and v_2 of `studyAt` edges e_1 and e_2, if e_1 and e_2 have the same value for property `semester`, then it must be the case that $v_1 = v_2$.

4.5 Validation of PG-Keys

The crucial task related to PG-Keys is *validation*; that is, determining if a given property graph satisfies a given PG-Key. Validation of PG-Keys can be recast as query evaluation. Indeed, recall that the

satisfaction of a PG-Key is expressed in terms of conditions **(K1)**, **(K2)**, and **(K3)**. Each of these conditions can be reformulated as emptiness of a query built from the scope and the descriptor of the PG-Key. We explain this with an example from Section 4.2:

```
FOR (x:City) IDENTIFIER y, x.name WITHIN
  (x)-[:isPartOf]->(y:Country).
```

The query for **(K1)** is obtained by combining two copies of the scope with different scope variables and two copies of the descriptor with the same descriptor variable:

```
MATCH (x1:City), (x1)-[:isPartOf]->(y:Country),
      (x2:City), (x2)-[:isPartOf]->(y:Country)
      WHERE x1 <> x2 AND x1.name IS NOT NULL AND
            x2.name IS NOT NULL AND x1.name = x2.name
RETURN x1, x2.
```

Because the property `name` is also a component of the key, we additionally check that it is set and that `x1.name = x2.name`. The resulting query finds pairs of different targets that share a key value. Hence, **(K1)** holds exactly when the query returns no answers.

For **(K2)**, we select targets in the scope for which the descriptor cannot be matched:

```
MATCH (x:City)
WHERE NOT EXISTS(x.name) OR
      NOT (x)-[:isPartOf]->(y:Country)
RETURN x.
```

Notice that matching the descriptor also involves checking that the property `name` is set. Again, **(K2)** holds exactly when this query returns no answers.

For **(K3)**, the query selects targets for which two different key values exist. It is built from one copy of the scope and two copies of the descriptor with different descriptor variables:

```
MATCH (x:City), (x)-[:isPartOf]->(y1:Country),
              (x)-[:isPartOf]->(y2:Country)
WHERE y1 <> y2
RETURN x.
```

Note that we do not need to check that there is only one value of the property `name`, because our data model does not include multi-valued properties. Like in both previous cases, **(K3)** holds exactly when the constructed query returns no answers.

Clearly, such rewritings into queries can be directly obtained for any PG-Key. This additionally addresses the design requirement R6 identified in Section 2, allowing to express the semantics of PG-Keys in the very familiar terms of query semantics. Moreover, while additional mechanisms would be needed to handle aspects like batching or incremental validation, implementations of PG-Keys can leverage existing facilities for efficient query evaluation. Hence, PG-Keys have excellent potential for direct deployment and impact in practice, satisfying design requirement R7.

Incremental validation. Database constraints must be enforced by a DBMS when the state of a database changes after an update. This is standard with relational constraints and updates. The state of graph database updates is far from being fixed, with GQL not yet offering such facilities, and even in well-established languages such as Cypher it is accepted that update features need to be redesigned as they have several deficiencies [38]. Some variants of PG-Keys are very close to relational (see Section 5.2) and thus easily maintainable under updates that resemble relational insertions/deletions. Others are more complex, akin to SQL's assertions, and thus their incremental validation will require techniques from

incremental view/integrity maintenance [13]. Such techniques are well developed for languages that do not use recursion or reachability queries, see e.g. [39, 69]. If reachability, or more generally regular path queries (see Section 6.2) are present, incremental validation with non-recursive queries becomes impossible without the use of complex auxiliary data structures [15, 30] and specialized algorithms based on maintenance of datalog queries [40, 61]. Hence, the cost of incremental validation will heavily depend on the cost of maintaining the underlying complex data structures under insertions, deletions and update operations. Related to this is the question of constraint-enforced cascading updates, similarly to cascading deletes in the presence of foreign keys. All the above issues need to be studied once the standardization process of graph query and update languages has concluded.

To summarize this section, we conclude that PG-Keys satisfies all eight of the design requirements (R0–R7) specified in Section 2.

5 RELATIONSHIP TO OTHER PARADIGMS

In this section, we compare PG-Keys to key formalisms in existing database models and data models.

5.1 Conceptual Data Models

By *conceptual data models* we mean here data models that are conceptual in nature, i.e., the Entity-Relationship Model, UML Class diagrams and ORM diagrams.

The Entity-Relationship Model. The classical ER Model [22] allows a group of attributes to be declared as *key* of an entity type. Moreover, it introduces the notion of *weak entity-type*, which has a *partial key* that, combined with the keys of entity types that are connected via *identifying relationships*, can identify entities in the entity type.

Both constructs are easily represented in PG-Keys, assuming that the nodes that represent entities are labeled with the entity type. For example, if the entity type Person has a key consisting of name and birthday then this can be represented as

```
FOR (x:Person) IDENTIFIER x.name, x.birthday.
```

The same holds for weak entity types with partial keys. For example, consider a case where we have a weak entity type City, which is identified by a combination of its attribute name and the entity of type Country that it is reached via the identifying relationship isPartOf. This can be represented in PG-Keys as

```
FOR (x:City) IDENTIFIER y, x.name WITHIN
  (x)-[:isPartOf]->(y:Country).
```

UML Class Diagrams. UML class diagrams model the structure of a system in terms of classes and their relationships. When used for the *conceptual perspective* [35] they can serve as a conceptual data model. There is no special graphical notation for keys, and these are usually represented by comments or stereotypes. However, OCL, the Object Constraint Language, which is part of UML, allows the expression of key constraints with the collection operator isUnique(). For example, if a class City represents a weak entity type where entities are identified by their name and the Country they are part of, then the EXCLUSIVE aspect of the key constraint can be represented as

```
City.allInstances ->
  isUnique(Tuple{ctyName=name,cntryName=isPartof}).
```

There is a close similarity to PG-Keys: the collection to which isUnique() is applied corresponds to the scope of a PG-Key, which in the example is defined by City.allInstances. The function with which isUnique() is parameterized, which is here a function that constructs a tuple containing the *name* property and the country that the city is part of, corresponds to the descriptor. Given this similarity, the expressive power of the isUnique() operator is similar to that of PG-Keys if the used OCL expressions for defining the scope and the descriptor are similar in expressive power to the queries used in the PG-Keys.

ORM Diagrams. The Object-Role Modelling (ORM) approach [42] extends Entity-Relationship modelling with an elaborate graphical notation for a wide range of constraints and a specific language-based design methodology. It treats attributes and relationships as equals and unifies them in the concept of *fact types*. As a consequence, it uses for each the same notation to denote key, cardinality and other constraints.

An example of an ORM diagram with key constraints is:
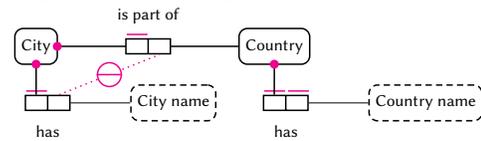
It illustrates an *internal uniqueness constraint* on the fact type is part of, represented by a line over its first role, which indicates that each City has at most one associated Country. It also illustrates an *external uniqueness constraint*, represented by a horizontal line in a circle connected to the second role of is part of and the second role of City.has.City name, which indicates that each combination of Country and City name is associated with at most one City. It will be clear that the internal uniqueness constraint corresponds to an EXCLUSIVE PG-Key for edges, and the external uniqueness constraint to an EXCLUSIVE PG-Key for nodes.

In addition ORM has the concept of *reference scheme* which is a special type of external uniqueness constraint that corresponds to the IDENTIFIER constraint in PG-Keys. This latter notion is explicitly designed as the mechanism that allows users to refer unambiguously to objects in the instance of a diagram [43].

As illustrated in the ORM diagram, the uniqueness constraints can be combined with mandatory participation constraints, indicated by bullets. This allows in fact the representation of all the four types of keys in PG-Keys.

5.2 Relationships to Relational Keys

Relational data are often migrated to graph databases, where expensive joins can be replaced by more efficient navigational exploration allowed by graph query languages. Can the key constraints present in a relational database R be expressed by PG-Keys over the corresponding graph representation G_R? We illustrate how this is possible by means of an example that involves both keys and foreign keys. Consider the relations:

```
City(ID,name,country,population),
Person(name,birthday,cityID)
```

storing information about persons and the city they live in. We assume that cityID is a foreign key referring to the ID attribute of City. Suppose that these relations are translated to a property

graph, as follows. Each tuple in `City` becomes a different node with label `City` and properties `ID`, `name`, `country`, and `population`. Each tuple in `Person` becomes a different node with label `Person` and properties `name` and `birthday`. Finally, for each tuple in `Person` with `cityID` not null, we add a `livesIn`-edge from the relevant person node to the city node that is identified by `cityID`. This translation follows closely that of a recent proposed method for systematically mapping relational data to property graphs [73].

To express that ID is a key for `City`, we use the PG-KEY

```
FOR (x:City) IDENTIFIER x.ID .
```

Notice that each node always has at most one value of the property `ID`, just like each tuple in the relation `City` has at most one value in the column `ID`. Consequently, the `SINGLETON` restriction, built into `IDENTIFIER`, is redundant here and we can equivalently use `EXCLUSIVE MANDATORY`. Candidate keys and uniqueness constraints, including multi-attribute ones, can be handled in a similar fashion. For instance, to express that (`name`,`country`) is unique in the table `City`, we can use

```
FOR (x:City) EXCLUSIVE x.name, x.country .
```

PG-KEYS can also ensure referential integrity on the property graph side. The foreign key itself expresses only that the entity in which a person lives is a city. However, assuming that the property graph will be allowed to evolve after translation, in order to guarantee that it can be translated back to relations we also need to ensure that each person lives in at most one place. (Recall that each `Person` node in the graph corresponds to a single *tuple* in the relation `Person`.) The combination of these two conditions can be expressed using the following two PG-KEYS:

```
FOR (x:Person) EXCLUSIVE SINGLETON y WITHIN
   (x)-[y:livesIn]->(),
FOR (x:Person)-[:livesIn]->() IDENTIFIER y WITHIN
   (x)-[y:livesIn]->(:City) .
```

The first PG-KEY states that, for each node with label `Person`, there is at most one outgoing `livesIn`-edge. Notice that this requires using `SINGLETON`, which does not have a counterpart in the relational setting. The second PG-KEY states that each node with label `Person` and an outgoing `livesIn`-edge, has exactly one outgoing `livesIn`-edge pointing to a node with label `City`. The precise formulation of referential integrity constraints on the graph side depends on the concrete translation from relations to property graphs. However, PG-KEYS are sufficiently flexible to express these constraints if the translations stay true to the underlying data.

Having taken care of the foreign key, we can easily handle keys built on top of it. For instance, to express that (`name`,`cityID`) is a key in the relation `Person` we can use PG-KEY

```
FOR (x:Person) IDENTIFIER x.name, y.ID WITHIN
   (x)-[:livesIn]->(y:City) .
```

Overall, under natural translations from relations to property graphs, PG-KEYS offer full support for relational keys and foreign keys. Moreover, while in the relational model the scope is always one whole relation and the descriptor is a list of attributes, in PG-KEYS the power to specify the scope and the descriptor is only limited by the chosen query language. The expressiveness of such keys (as those of object-oriented databases [21]) is similar to those in the ER Model in that it allows the representation of strong entity types and weak entity types. In general, a formal comparison with relational constraints is an important topic for further study.

5.3 XML Keys

The literature on XML keys includes industrial standards like XML Schema [53] and DSDL (including RelaxNG and Schematron) [1], as well as a large body of academic work [18–20, 44, 54] proposing various extensions and improvements. Due to the space limitations we focus on XML Schema.

XML Schema offers two constructors for keys: `UNIQUE` and `KEY`. `UNIQUE` requires that an attribute or element value must be unique within a certain scope. For instance,

```
<unique name="PersonUnique">
  <selector xpath="../Person"/>
  <field xpath="@name"/>
  <field xpath="@birthday"/>
</unique>
```

corresponds to the `EXCLUSIVE SINGLETON` example in Section 5.4. The selector describes the scope: every `Person` node in the tree. The fields specify the components of the key: the values of the attributes `@name` and `@birthday` of each selected node. The XPath expression in the field must return a single value for each selected node.

`KEY` extends `UNIQUE` such that an entity value must be unique and cannot be set to nil (i.e., is not nillable) which corresponds to our `IDENTIFIER` when focusing on values. For instance,

```
<key name="personKey">
  <selector xpath="../Person"/>
  <field xpath="@email"/>
</key>
```

corresponds to the PG-KEY

```
FOR (x:Person) IDENTIFIER x.email .
```

As we can see, there is indeed a close relationship between keys for XML and PG-KEYS. The main difference between the two is perhaps the *navigational language*. Whereas XPath is perfectly suited for navigation in trees (and can be adapted for graphs [50]), we aim at using languages that were specifically designed for navigation in property graphs, such as GQL and Cypher.

5.4 Keys in Semantic Web Stack

We now discuss how PG-KEYS can be used to simulate key constraints available in the Semantic Web Stack (SWS). Due to the limited space, we forsake the existing academic work [49, 66] and focus on the standards [11, 46, 48, 71].

At the lower layers of SWS, RDF [48, 80] uses Internationalized Resource Identifiers (IRIs) to provide a rudimentary mechanism for reference but not identification since the same real-world object may be described with multiple IRIs.

Going up the SWS, OWL [11] supports keys using the `HasKey` construct. For example, assuming `:Person` is a class and `:name` and `:birthday` are properties, the snippet

```
:Person owl:hasKey (:name :birthday) .
```

asserts that instances of `:Person` are uniquely identified by the combination of the values of `:name` and `:birthday`; it does not say that those values exist nor that there is only one of each. This corresponds precisely to the PG-KEY

```
FOR (x:Person) EXCLUSIVE x.name, x.birthday .
```

If `:name` and `:birthday` were declared as functional, e.g.,

```
:name rdf:type owl:FunctionalProperty .
```

the corresponding PG-Key would be an `EXCLUSIVE SINGLETON` constraint. Ensuring that all key components exist, under the open-world assumption adopted by OWL, is very hard.

Recent RDF constraint languages [79], such as SHACL [26, 46] or ShEx [72], adopt the closed-world assumption. They do not have built-in key constraints, but they support cardinality constraints which can be used to emulate simple keys; keys with multiple components, like in the first example, cannot be expressed directly.

6 EXTENSIONS OF PG-KEYS

We discuss extensions of PG-Keys that are not in our core formalism but would need to be eventually supported by a fully fledged design.

6.1 NULL Values

So far we have tacitly assumed that property values cannot be nulls. In real life, nulls are abundant and arise for two principal reasons. First, a value may exist but be currently unknown (for example, the birthday of a person may not be known). Second, a value may not even exist (for example, one may refer to a property of a node that does not exist, say `x.age` instead of `x.birthday`). In relational database practice, and in particular in SQL, these different scenarios are represented by the same NULL [28], and the practice has been extended to graph query languages like Cypher [36, 38].

SQL's approach to handling nulls is based on a *three-valued logic (3VL)* that extends the standard Boolean logic of *true* and *false* with a truth value *unknown*. It can be summarized as follows: (1) every condition involving a null evaluates to *unknown*; (2) truth values propagate through connectives `AND`, `OR`, and `NOT` by using the rules of SQL's 3VL; (3) once a condition is evaluated in the `WHERE` clause of a SQL query, only *true* tuples remain.

With PG-Keys, we propose to follow SQL's approach that a constraint holds if it does not evaluate to *false*. In this case one can validate a key θ by a query Q_θ that looks for violations of θ, that is, it computes witnesses of the negation of θ, as is explained in Section 4.5. If the key θ itself evaluates to *true* or *unknown*, then its negation is *false* or *unknown*, according to 3VL, and therefore Q_θ produces no output.

Nulls in conditions however need to be handled with care. To give a simple relational example (easily mimicked in a property graph), if we have a relation `R(A,B)` with a `UNIQUE` declaration on attribute `A`, then adding tuples `(NULL,1)` and `(NULL,2)` is possible. The uniqueness constraint, stating that there are no two different tuples with the same value of `A`, will evaluate to *unknown*, and hence will be validated. To fall back to the two-valued logic of *true* and *false*, one needs to impose `NOT NULL` constraints. This is what happens with primary keys: then the situation above when two NULLs can be entered is no longer possible.

We now explain how PG-Keys work in the presence of nulls. To start, we must specify what the queries used in PG-Keys are. We assume, in line with all the examples so far, that they are patterns with further constraints on property values, for example,

```
x WITHIN (x:Person) WHERE x.age > 30 .
```

With each such query $q(\bar{x})$, we associate a new query $q_{\text{notnull}}(\bar{x})$ which is the same as q but with added `IS NOT NULL` conditions for each property value used in the query. For example, the above query would be transformed into

```
x WITHIN (x:Person) WHERE x.age>30 AND x.age IS NOT NULL .
```

Such a query rules out both kinds of nulls, be they due to unknown value of `age` or to the absence of the property `age` altogether.

For each condition **(Ki)**, for $i = 1, 2, 3$, used in the definition of satisfaction of PG-Keys in Section 4.1, we define a condition **(Ki)$_{\text{null}}$** which is the same as **(Ki)** except that $p(x)$ is replaced by $p_{\text{notnull}}(x)$ and $q(x, \bar{y})$ by $q_{\text{notnull}}(x, \bar{y})$. Notice that it is *easier* to satisfy **(K1)$_{\text{null}}$** and **(K3)$_{\text{null}}$** than **(K1)** and **(K3)** respectively, because only null-free objects must be looked at. More precisely, all the `IS NOT NULL` conditions appear in the antecedent, and since it is harder to satisfy the antecedent, it is thus easier to satisfy the whole constraint. In fact, **(Ki)**, for $i = 1, 3$, holds (i.e., does not evaluate to *false*) in 3VL if and only if the condition **(Ki)$_{\text{null}}$** is true. In other words, to check satisfaction of `EXCLUSIVE` and `EXCLUSIVE SINGLETON` we can disregard nulls.

The situation with `MANDATORY`, i.e., **(K2)** is more involved, since **(K2)$_{\text{null}}$** captures our intuition of such constraints, but disagrees with the 3VL semantics of **(K2)**. To illustrate this, consider

```
FOR (x:Person) EXCLUSIVE MANDATORY y WITHIN
    (x)-[:owns]->(y:Passport) WHERE y.expiry > $today .
```

Suppose that we have a single person in the database whose passport expiry date is NULL. Condition **(K2)** defining MANDATORY constraints in this case evaluates to *unknown*, and thus in the SQL-inspired approach the constraint is satisfied. Condition **(K2)$_{\text{null}}$**, on the other hand, is false. In the case of `MANDATORY`, this is the desired behavior and indeed mandatory keys should avoid nulls, similarly to primary keys in SQL that mandate `NOT NULL` for attributes involved.

One way of resolving this is to assume that conditions involving NULL evaluate to *false* rather than *unknown*. The idea in itself is not new, it was present in old query languages such as Quel, and was recently studied in connection with various proposals on using 2-valued logic in place of SQL's 3VL, see [24, 25]. In such a logic, the above MANDATORY condition would *not* hold for a person whose passport expiry is NULL, thus fulfilling our intuition about these constraints. At the same time, it does not affect other constraints. In fact, under this 2-valued logic of nulls, condition **(Ki)** is true iff **(Ki)$_{\text{null}}$** is true, for $i = 1, 2, 3$. We would thus advocate one of two options for handling nulls in PG-Keys: either use a 2-valued logic as explained here, or follow SQL's 3VL but then introduce `NOT NULL` declarations for properties used in `MANDATORY` constraints.

6.2 Regular Path Queries

In Section 3, we assumed that key constraints for property graphs are defined using queries of the form $q(\bar{x})$, where $\bar{x}$ is a tuple of variables that bind to *nodes*, *edges*, and *property values*. We now extend $\bar{x}$ to variables that bind to *paths*, the latter being first-class citizens in several graph query languages [6, 36]. In the research literature [7, 12] and in practical query languages, this is usually done by incorporating *regular path queries (RPQs)* or a subset thereof. (The term RPQ is fairly standard in the research literature, but has diverse names in practical languages, e.g., *property paths* or *path patterns*.)

As an example, let us assume that the `isPartOf` relation in Figure 2 is extended (as in LDBC Social Network Benchmark [5, 32]) to a larger hierarchy of geographical entities, containing provinces and

continents. The following PG-Key expresses that every isPartOf path from a city to a continent can be uniquely identified by the city and the continent:

```
FOR p WITHIN p = (:City)-[:isPartOf*]->(:Continent)
IDENTIFIER x, z WITHIN
  p = (x:City)-[:isPartOf*]->(z:Continent) .
```

The variable p binds to a path in both the scope and the descriptor of the constraint (we use a GQL-like syntax for path variables).

The following example shows the usage of paths as key components: the PG-Key

```
FOR (x:City) EXCLUSIVE x.name, p WITHIN
  p = (x)-[:isPartOf*]->(:Country)
```

asserts that if two cities within a country have the same name, then their paths to the country must differ (e.g., go via different states).

Both constraints use the transitive operator isPartOf*, which matches to paths of arbitrary length in which each edge is labeled isPartOf. Notice that such use of isPartOf makes the constraints robust against changes in the data. For instance, if the geographical hierarchy would be extended in the data, e.g., by adding *counties* or *states*, then the above mentioned constraints do not need to be updated to reflect these changes in the data.

Transitive operators in key constraints also pose challenges, however. One challenge is how they deal with cyclicity in the data. This particular challenge also arises with the corresponding operators in query languages, and is part of an ongoing discussion. Essentially, the question is: which paths do we allow to match against these expressions? Current systems are working with four different variants: unrestricted, no repeated nodes (simple paths), no repeated edges (trails), or shortest paths [6, 36].

The choice between these alternatives is not entirely trivial, because it may have a large influence on the complexity of evaluation. In the unrestricted case, RPQs can be evaluated in polynomial time, whereas deciding whether a pair of nodes in V is in the result set of an RPQ under simple path or trail semantics is NP-complete in general. The latter becomes tractable for certain fixed regular path expressions, leading to the classes C_{tract} [10] and T_{tract} [55] highly frequent in real-world query logs [16, 17]. The class T_{tract} precisely identifies the set of regular expressions for which the data complexity under trail semantics is tractable if P ≠ NP, and the slightly smaller class C_{tract} does the same for the simple path semantics. An even more restricted class included in C_{tract} and consisting of *simple transitive expressions* [56] leads to only use simple subexpressions of the kind a^* also very common in practice [16]. The query in the scope of the second constraint above falls in this latter class.

It is currently not clear which of these semantics is preferable in practice. Whereas the arbitrary path semantics has a lower evaluation complexity [63], simple path and trail semantics avoid issues with infinitely many results. Furthermore, arbitrary path semantics may lead to some issues when used for evaluating the queries within key constraints. For example, if a constraint required a path to be unique, then this would mean that this path cannot have a cycle, as cycles can be repeatedly traversed. Thus, simple path or trail semantics might be preferable in this case, bringing up the question of which semantics to use to evaluate key constraints, which is an interesting direction of investigation for next-generation graph databases implementing these constraints.

6.3 Complex Values

In our data model we allow complex property values, such as tuples, sets, lists, or even arbitrary JSON structures, but so far we have been treating them atomically. In order to support them fully, we need to navigate inside their complex structure.

Suppose that university students can be identified by each of their multiple official email addresses. Furthermore, assume that the addresses are stored in an email property, organized into a JSON tree structure, listing addresses together with their categories (official/unofficial). To express that *each* official address is unique, we can use for instance JSONPath [37] to iterate over the list and, for each entry storing an official address, recover the actual value of the address:

```
FOR x WITHIN (x:Person)-[:studyAt]->(:University)
EXCLUSIVE MANDATORY x.email[@.category='official'].address .
```

Analogously to selecting simple values from a complex value, it is also possible to use descriptors to collect multiple simple values into one complex value. For instance, to express that an entity is uniquely defined by an (unordered) collection of values, we can deploy a descriptor that collects all these values into a set, and define a key with this set as a component.

7 LOOKING AHEAD

The LDBC Property Graph Schema Working Group has reached an important consensus and milestone by producing this recommendation for the design of property graph key constraints.

Our recommendation must be framed in its broader context: the design of the new GQL graph query language by the ISO/IEC JTC1 SC32 WG3 committee. PG-Keys is informing the design of GQL via the LDBC liaison, by having the main elements of this proposal incorporated into its future standard. Furthermore, our recommendation extends the expressivity of property graphs which may help facilitate mapping to other data models.

This paper is a call to action for industry and academia driving the graph database industry. From a developer standpoint, PG-Keys can influence the implementation of property graph keys in commercial and non-commercial graph database systems as well as their applications in many graph processing tasks. On the research side, our framework triggers a number of open problems that can attract the attention of the data management community. These problems include the *validation and maintenance complexity* of PG-Keys for specific query languages, the *implication and inference problems*, as well as the use of *static analysis for optimization purposes*. Furthermore, the handling of *nulls* in PG-Keys suggests alternatives to SQL's three-valued logic that need to be further explored.

Finally, the work of the PGSWG is not done. We are in the process of establishing consensus with respect to the semantics of a schema language and to extensions of the property graph data model to support features such as meta-properties.

ACKNOWLEDGMENTS

REFERENCES

[1] ISO/IEC 19757. 2016. *Information technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation — Schematron.* Standard. International Organization for Standardization, Geneva, CH.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[3] AgensGraph. 2020. AgensGraph. https://bitnine.net/agensgraph (visited: 2020-11).

[4] Amazon. 2020. Amazon Neptune. https://aws.amazon.com/neptune/ (visited: 2020-11).

[5] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep-Lluís Larriba-Pey, and Norbert Martínez-Bazan et al. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020).

[6] 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.

[7] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *Comput. Surveys* 50, 5 (2017), 68:1–68:40.

[8] Renzo Angles and Claudio Gutiérrez. 2008. Survey of graph database models. *Comput. Surveys* 40, 1 (2008), 1:1–1:39.

[9] Apache. 2020. TinkerPop. https://tinkerpop.apache.org/ (visited: 2020-11).

[10] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2020. A trichotomy for regular simple path queries on graphs. *J. Comput. Syst. Sci.* 108 (2020), 29–48.

[11] Jie Bao, Deborah McGuinness, Elisa Kendall, and Peter Patel-Schneider. 2012. *OWL 2 Web Ontology Language Quick Reference Guide (Second Edition).* W3C Recommendation. W3C. https://www.w3.org/TR/2012/REC-owl2-quick-reference-20121211/.

[12] Angela Bonifati and Stefania Dumbrava. 2018. Graph Queries: From Theory to Practice. *SIGMOD Rec.* 47, 4 (2018), 5–16.

[13] Angela Bonifati, Stefania Dumbrava, and Emilio Jesús Gallego Arias. 2018. Certified Graph View Maintenance with Regular Datalog. *Theory Pract. Log. Program.* 18, 3-4 (2018), 372–389.

[14] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs.* Morgan & Claypool Publishers.

[15] Angela Bonifati, Martin Hugh Goodfellow, Ioana Manolescu, and Domenica Sileo. 2013. Algebraic incremental maintenance of XML views. *ACM Trans. Database Syst.* 38, 3 (2013), 14:1–14:45.

[16] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the Maze of Wikidata Query Logs. In *WWW.* ACM, 127–138.

[17] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679.

[18] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. 2002. Keys for XML. *Computer Networks* 39, 5 (2002), 473–487.

[19] Peter Buneman, Susan B. Davidson, Wenfei Fan, Carmem S. Hara, and Wang Chiew Tan. 2003. Reasoning about keys for XML. *Information Systems* 28, 8 (2003), 1037–1063.

[20] Peter Buneman, Wenfei Fan, Jérôme Siméon, and Scott Weinstein. 2001. Constraints for Semi-structured Data and XML. *SIGMOD Rec.* 30, 1 (2001), 47–45.

[21] R. G. G. Cattell and Douglas K. Barry. 2000. *The Object Data Standard: ODMG 3.0.* Morgan Kaufmann.

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

[23] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. *Entity Resolution in the Web of Data.* Morgan & Claypool Publishers.

[24] Marco Console, Paolo Guagliardo, and Leonid Libkin. 2018. Propositional and Predicate Logics of Incomplete Information. In *KR.* AAAI Press, 592–601.

[25] Marco Console, Paolo Guagliardo, Leonid Libkin, and Etienne Toussaint. 2020. Coping with Incomplete Data: Recent Advances. In *PODS.* ACM, 33–47.

[26] Julien Corman, Juan L. Reutter, and Ognjen Savkovic. 2018. Semantics and Validation of Recursive SHACL. In *International Semantic Web Conference (1) (Lecture Notes in Computer Science, Vol. 11136).* Springer, 318–336.

[27] DataStax. 2020. DataStax. https://datastax.com/ (visited: 2020-11).

[28] C. J. Date and H. Darwen. 1996. *A Guide to the SQL Standard.* Addison-Wesley.

[29] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *SIGMOD Conference.* ACM, 377–392.

[30] Guozhu Dong, Leonid Libkin, and Limsoon Wong. 2003. Incremental recomputation in local languages. *Inf. Comput.* 181, 2 (2003), 88–98.

[31] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *IEEE Trans. Knowl. Data Eng.* 19, 1 (2007), 1–16.

[32] Orri Erling, Alex Averbuch, Josep-Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *SIGMOD Conference.* ACM, 619–630.

[33] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. 2015. Keys for Graphs. *Proc. VLDB Endow.* 8, 12 (2015), 1590–1601.

[34] Wenfei Fan and Ping Lu. 2019. Dependencies for Graphs. *ACM Trans. Database Syst.* 44, 2 (2019), 5:1–5:40.

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

[36] 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.

[37] Stefan Gössner. 2007. JSONPath. https://goessner.net/articles/jsonpath/ (visited: 2021-02).

[38] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. 2019. Updating Graph Databases with Cypher. *Proc. VLDB Endow.* 12, 12 (2019), 2242–2253.

[39] Timothy Griffin and Leonid Libkin. 1995. Incremental Maintenance of Views with Duplicates. In *SIGMOD.* ACM Press, 328–339.

[40] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *SIGMOD.* 157–166.

[41] Claudio Gutierrez and Juan F. Sequeda. 2020. Knowledge Graphs: A Tutorial on the History of Knowledge Graph's Main Ideas. In *CIKM Conference.* ACM, 3509–3510.

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

[43] Terry A. Halpin. 2013. Modeling of Reference Schemes. In *BMMDS/EMMSAD (Lecture Notes in Business Information Processing, Vol. 147).* Springer, 308–323.

[44] Sven Hartmann and Sebastian Link. 2009. Expressive, yet tractable XML keys. In *EDBT (ACM International Conference Proceeding Series, Vol. 360).* ACM, 357–367.

[45] JanusGraph. 2020. JanusGraph. https://janusgraph.org/ (visited: 2020-11).

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

[47] Redis Labs. 2020. RedisGraph. https://redislabs.com/modules/redis-graph/ (visited: 2020-11).

[48] Markus Lanthaler, David Wood, and Richard Cyganiak. 2014. *RDF 1.1 Concepts and Abstract Syntax.* W3C Recommendation. W3C. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[49] Georg Lausen. 2007. Relational databases in RDF: Keys and foreign keys. In *Semantic Web, Ontologies and Databases.* Springer, 43–56.

[50] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. 2016. Querying Graphs with Data. *J. ACM* 63, 2 (2016), 14:1–14:53.

[51] Sebastian Link. 2020. Neo4j Keys. In *ER (Lecture Notes in Computer Science, Vol. 12400).* Springer, 19–33.

[52] Artem Lysenko, Irina A. Roznovat, Mansoor Saqi, Alexander Mazein, Christopher J. Rawlings, and Charles Auffray. 2016. Representing and querying disease networks using graph databases. *BioData Min.* 9 (2016), 23.

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

[54] Wim Martens, Frank Neven, Matthias Niewerth, and Thomas Schwentick. 2017. BonXai: Combining the Simplicity of DTD with the Expressiveness of XML Schema. *ACM Trans. Database Syst.* 42, 3 (2017), 15:1–15:42.

[55] Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. A Trichotomy for Regular Trail Queries. In *STACS (LIPIcs, Vol. 154).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:16.

[56] Wim Martens and Tina Trautner. 2019. Dichotomies for Evaluating Simple Regular Path Queries. *ACM Trans. Database Syst.* 44, 4 (2019), 16:1–16:46.

[57] MemGraph. 2020. MemGraph. https://memgraph.com/ (visited: 2020-11).

[58] Antonio Messina, Antonino Fiannaca, Laura La Paglia, Massimo La Rosa, and Alfonso Urso. 2018. BioGraph: a web application and a graph database for querying and analyzing bioinformatics resources. *BMC systems biology* 12, 5 (2018), 98.

[59] Microsoft. 2020. Azure Cosmos. https://azure.microsoft.com/ (visited: 2020-11).

[60] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[61] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. 2019. Maintenance of datalog materialisations revisited. *Artif. Intell.* 269 (2019), 76–136.

[62] Oracle. 2020. Oracle Spatial and Graph. https://www.oracle.com/database/technologies/spatialandgraph.html (visited: 2020-11).

[63] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2020. Regular Path Query Evaluation on Streaming Graphs. In *SIGMOD Conference.* ACM, 1415–1430.

[64] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications. In *GRADES@SIGMOD/PODS.* ACM, 12:1–12:7.

[65] M. P. Papazoglou, S. Spaccapietra, and Z. Tari. 2000. *Identifying Objects by Declarative Queries*. 255–277.
[66] Jan Paredaens. 2012. What about Constraints in RDF? In *Conceptual Modelling and Its Theoretical Foundations*. Springer, 7–18.
[67] Norman W. Paton and Peter M. D. Gray. 1988. Identification of Database Objects by Key. In *OODBS (Lecture Notes in Computer Science, Vol. 334)*. Springer, 280–285.
[68] Jaroslav Pokorný, Michal Valenta, and Jiří Kovačič. 2017. Integrity constraints in graph databases. *Procedia Computer Science* 109 (2017), 975–981.
[69] Xiaolei Qian and Gio Wiederhold. 1991. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.* 3, 3 (1991), 337–341.
[70] Ian Robinson, Jim Webber, and Emil Eifrem. 2013. *Graph databases*. O'Reilly Media.
[71] Guus Schreiber and Mike Dean. 2004. *OWL Web Ontology Language Reference*. W3C Recommendation. W3C. https://www.w3.org/TR/2004/REC-owl-ref-20040210/.
[72] Sławek Staworko, Iovka Boneva, José Emilio Labra Gayo, Samuel Hym, Eric G. Prud'hommeaux, and Harold R. Solbrig. 2015. Complexity and Expressiveness of ShEx for RDF. In *ICDT (LIPIcs, Vol. 31)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 195–211.
[73] Radu Stoica, George Fletcher, and Juan F. Sequeda. 2019. On Directly Mapping Relational Databases to Property Graphs. In *AMW*.
[74] Neil Swainston, Riza Batista-Navarro, Pablo Carbonell, Paul D. Dobson, Mark Dunstan, Adrian J. Jervis, Maria Vinaixa, Alan R. Williams, Sophia Ananiadou, Jean-Loup Faulon, Pedro Mendes, Douglas B. Kell, Nigel S. Scrutton, and Rainer Breitling. 2017. biochem4j: Integrated and extensible biochemical knowledge through graph databases. *PLOS ONE* 12, 7 (07 2017), 1–14.
[75] Sparsity Technologies. 2020. Sparksee. https://sparsity-technologies.com/#sparksee (visited: 2020-11).
[76] Neo Technology. 2020. Neo4j. https://neo4j.com/ (visited: 2020-11).
[77] TigerGraph. 2020. TigerGraph. https://www.tigergraph.com/ (visited: 2020-11).
[78] Titan. 2020. Titan. https://titan.thinkaurelius.com/ (visited: 2020-11).
[79] Dominik Tomaszuk. 2017. RDF validation: A brief survey. In *International Conference: Beyond Databases, Architectures and Structures*. Springer, 344–355.
[80] Dominik Tomaszuk and David Hyland-Wood. 2020. RDF 1.1: Knowledge representation and data integration language for the Web. *Symmetry* 12, 1 (2020), 84.
[81] Yuhang Xia and Chenglin Sun. 2018. Property Graph Database Modeling and Application of Electronic Medical Record. In *IMCCC Conference*. IEEE, 963–967.
[82] Zuopeng Justin Zhang. 2017. Graph databases for knowledge management. *IT Professional* 19, 6 (2017), 26–32.