OAEP-2023-04 **DOI: 10.54285/ldbc.OFJF3566**

Published on LDBC's website April 2023

Originally published privately for members of ISO/IEC JTC 1/SC32 WG3, and of the LDBC PGSWG, June 2020

LDBC Property Graph Schema contributions to WG3

LDBC Property Graph Schema Working Group (PGSWG) contributions to WG3 (MMX) June 2020

The papers listed below were submitted to WG3 for discussion at its MMX (Malmö, Sweden) meeting in June 2020.

This document contains five original papers, reproduced without substantive modification. Three sets of accompanying slides produced for presentation and discussion at WG3 are included in an Appendix.

The titles given here are the titles under which they were uploaded to the **ISO/IEC JTC1/SC32 WG3** document register, which differ slightly in some cases from the internal titles in the documents themselves.

1	**MMX-069**	"GS-Basic Overview report"	Ed Jan Hidders. Multiple authors.	12 June 2020
2	**MMX-070**	"GS-Basic Formal definitions report"	Ed. Jan Hidders, Juan Sequeda. Author Dominik Tomaszuk.	12 June 2020
3	**MMX-072r1**	"Meta-properties"	Ed. Bei Li. Multiple authors.	June 2020
4	**MMX-074**	"PGS-Keys report"	Ed. George Fletcher. Multiple authors.	June 2020
5	**MMX-076**	"Proposal for requirements scope and roadmap"	Alastair Green	16 June 2020

Appendix		Slides for presentation and discussion of the preceding papers at WG3		
A1	**MMX-071**	“GS-Basic”	Jan Hidders	24 June 2020
A2	**MMX-073r1**	“Meta-properties overview”	Bei Li	4 June 2020
A3	**MMX-075**	“Key Constraints”	LDBC PGSWG	15 June 2020

LDBC Open Access to External Papers

In this series, Linked Data Benchmark Council makes papers published originally for a restricted audience available for open access.

Such papers are of interest to our members and the public, and are concerned with topics that relate to the work of LDBC. They are published with the permission of their copyright holders, which may have been given by a licence grant.

This collection of eight papers is relevant for the work of the LEX (LDBC Extended GQL Schema) Working Group.

These papers have the character of technical reports: they have not been submitted to or accepted via peer review by an established scholarly publication.

LDBC PGS-B:BAS-01r1

GS-Basic Overview report

Editor: Jan Hidders
Authors: Juan Sequeda, Dominik Tomaszuk, Alastair Green, Victor Marsault, Keith W. Hare, Victor Lee

References

[GQL:DATA MODEL] ISO/IEC JTC1 SC32 WG3 SXM-030r3 *"The GQL property graph data model "*

[GQL:EWD] ISO/IEC JTC1 SC32 WG3 WG3:MMX-010r2 *"GQL Early Working Draft V4.2"*

[LDBC PGS:JH-03r7] *On the syntax and semantics of GS-Basic without inheritance*, Eds. Jan Hidders and Juan Sequeda

[LDBC PGS-B:BAS-02] *GS-Basic Formal definitions report*, Ed. Jan Hidders and Juan Sequeda

[NEO4J:DOC] The Neo4j Getting Started Guide v4.0

[TG:DOC] TigerGraph Documentation

[DSE:G] DataStax Enterprise Graph

[TP] Apache Tinkerpop

[JG] JanusGraph

[COSMOS] Azure Cosmos DB documentation

[NEPTUNE] Amazone Neptune

[OMG:UML] OMG Unified Modelling Language, version 2.5.1, December 2017, Object Management Group.

[ISO:UML] ISO/IEC 19505-1:2012 Information technology — Object Management Group Unified Modeling Language (OMG UML) — Part 1: Infrastructure

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

[Halpin:2015] Halpin, T. (2015). *Object-Role Modeling Fundamentals: A Practical Guide to Data Modeling with ORM* (First edition). Technics Publications.

[Basic:23-03-2020] GS-Basic: Progress report: 23 March 2020

[LDBC PGS:AG-05] GS-Basic—Type conformance DRAFT 0.2

[LDBC PGS:AG-10r1] GS-Basic: Type/instance matching

[Bonifati:2019] Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., Voigt, H.: *Schema validation and evolution for graph databases*. In: Proceedings of the 38th International Conference on Conceptual Modeling (ER) (2019)

[LDBC PGS:AG-08] GS-Basic “Options 1 and 3”

Goal of this document

This document reports on the results of past discussions in the GS-Basic subgroup of the PGSWG working group. It focuses on the discussions on the topic of the syntax and semantics of the basic graph schemas for property graphs. It contains the results of these discussions as collected in [LDBC PGS:JH-03r7] and consisting of several proposals, their motivations and their trade-offs.

Executive Summary

This document discusses the syntax and semantics of graph-type cores, which consist of a set of vertex and edge types. It focuses on simple types, which means here that they are closed and describe only mandatory attributes. It also leaves inheritance as a topic for future discussions.

The semantics of graph-type core defines a conformance relationship between property graphs and a graph-type core. We distinguish two types of conformance: **strict conformance** and **weak conformance** (which we will also refer to as just conformance). The strict conformance is meant to capture the usual expectations for a database schema in that it exhaustively defines what data is expected or authorized to appear. It is designed so that one may predict how much memory is required to store vertices and edges. Intuitively, a subtype does not usually strictly conform to a supertype because one cannot predict how much more memory is required for storing the subtype. The weak conformance is meant to capture the expectations of a database schema as a type that specifies the minimal conditions that the input of a graph query or graph processing program must satisfy to prevent certain runtime errors. This means that it is defined such that each graph that conforms to a subtype also conforms to a supertype.

In this report several proposals for strict conformance and weak conformance are presented. The main proposal that has the most support within the GS-Basic group simply requires that for each element in the property graph there must be a matching type in the graph-type core. The other proposals are variants of this that attempt to add more expressive power or provide a more intuitive semantics. For a brief overview of all the proposals see Section Overview of proposals.

In this document the emphasis is on presenting the proposals in an informal manner. For the full formal definitions the reader is referred to the companion report [LDBC PGS-B:BAS-02]. That document contains also some additional material, such as:

- a fully described normalization procedure for the concrete syntax,
- a mapping between the at-least-one-match semantics and the combinatorial semantics,
- an extension of the at-least-one-match semantics for dealing with optional attributes and

- an in-depth analysis of the assumptions concerning the type system that governs attribute values.

Introduction

The focus of the GS-Basic group and this report is on the basic constructs in graph schemas (also referred to as *graph types* in [GQL:DATA MODEL]), which means here specifically the vertex and edge types. Moreover, the group has discussed the syntax and semantics of graph-type cores, by which we mean here the part of the graph schema that defines the set of element types that specify what type of elements are allowed in its instances. This means that the topics of schema constraints (additional constraints on what is allowed in instances of the schema, such as key constraints) and attribute types were considered initially out of scope. However, the semantics of attribute types turned out to be very closely connected to the semantics of graph-type cores, and so the group has studied the proposed attribute types and their semantics in [GQL:EWD]. This has resulted in some observations concerning the semantics of attribute types that are discussed in the companion of this report [LDBC PGS-B:BAS-02], which presents and discusses the formal definitions.

The scope of the discussions, and therefore also of this report, was for practical reasons limited to closed schemas and closed types. By “closed” we mean here that every vertex, edges and attribute must be somehow justified by the type or schema. This does not mean that the group thinks that types and schemas with a certain degree of openness are not important, quite the contrary, but just that this is a practical starting point for the discussion. In fact, the group has explored briefly some variants of open types to verify if the current proposals for closed types and schemas do not prevent an intuitive extension in that direction.

One specific type of openness was explicitly addressed by the group, namely the openness that is defined by allowing subtyping i.e., defining a semantics where an instance is in the extension of a type if it is in the extensions of a subtype. This means that we consider two types of semantics for element types and schemas. The first type, which we will call *strict conformance*, explicitly forbids undeclared subtyping. The second one does take subtyping into account and is called *weak conformance*, or also just *conformance*. This distinction between strict conformance and conformance is analogous to the one that is made in [GQL:EWD]. The reasons for explicitly including subtyping in the discussions was that its understanding played an important role in the debate on the most appropriate closed semantics of graph-type cores.

A final restriction in scope was that inheritance between element types would not be considered. Also this was because of practical reasons: some members felt that adding this would be straightforward and some felt it might be a serious complication and a contentious issue. Therefore it was decided to postpone its treatment.

Requirements

Although there was not an explicit list of precise requirements or use-cases that the schema language fragment that was under consideration should meet, there were several principles that were agreed should guide the discussions:

- **Compatibility with current ideas in the WG3 group.** The proposals cannot deviate too much from, or be deeply in conflict with, what is being considered by the WG3 group.
- **Well-understood relationships with schema languages in existing graph databases and graph computing frameworks.** The proposal cannot be too different from existing implemented schema languages in systems such as *Neo4j* [Neo4j:DOC], *TigerGraph* [TG:DOC], *DataStax Enterprise Graph* [DSE:GRAPH], *Tinkerpop* [TP], *JanusGraph* [JG], Azure Cosmos DB [COSMOS] and Amazon Neptune [NEPTUNE]. This does not mean that we intentionally try to define a common superset of all these languages, or a common subset, but we do try to stay in the spirit of these languages so that translations between schemas in the different languages are well understood.
- **Closeness to conceptual data modelling techniques**. Graph-based data models can be seen as an approximation of conceptual data modelling notations such as UML Class diagrams [OMG:UML] [ISO:UML], Enhanced ER diagrams [Elmasri:2015] and ORM diagrams [Halpin:2015]. Typical diagrams from such models should be straightforwardly representable in the schema language.
- **Preparedness for inheritance and openness.** The syntax and semantics of the considered fragment of the schema language should allow for a straightforward extension with inheritance and types with a more open nature.

Overview of proposals

In this report the following proposals are presented for the semantics:

- The **main proposal** which has the widest support and defines the semantics by requiring that for each element in the graph there must be at least one matching type in the graph-type core.
- The **combinatorial semantics** is more permissive and requires that for each element there is a combination of types in the graph-type core that justifies it.
- The **exactly-one-match semantics** requires that each element matches exactly one type in the graph-type core.
- The i**solation-aware semantics** assumes that vertices are already justified if they are incident to an edge that is justified. It therefore requires only for isolated vertices that a matching vertex type exists in the graph-type core.
- The **homomorphism-based semantics** interprets the graph-type core as a graph and requires that there exists a homomorphism from the property graph to this graph-type core graph. For these semantics also an isolation-aware variant is presented.

Detailed description of the proposals

In this section we describe in detail the different proposals. For the full formal definitions the reader is referred to [LDBC PGS-B:BAS-02].

Basic definitions

We start with establishing some standard terminology that we will use in the following sections. We will assume that there is a notion of **attribute name** (which generalises over labels and property names) and **attribute value**. We distinguish a special attribute value, `lbl`, that indicates that the attribute with this value represents a label.

Property graphs

A key notion is of course that of property graphs, which we will assume to be the following.

> **Definition:** A **property graph** is defined as consisting of (1) a set of vertices such that each vertex has some associated vertex content, (2) a set of edges such that each edge has an associated (a) tail vertex, (b) edge content and (c) head vertex. The tail and head vertices of each edge must be in the set of vertices, and the content that is associated with vertices and edges is a finite record that maps attribute names to attribute values.
>
> Both vertices and edges are assumed to be represented by an abstract identity, and so it is possible that a graph contains two vertices with the same content, and two edges that have the same tail vertex, head vertex and content.

As usual we will refer to the vertices and edges in a property graph as the **elements** of the graph.

Examples

Consider the following Property Graph

```
(v1 :Person {name = ”Jan Hidders”})
(v2 :City {name = “London”})
(v3 :City {name = “Brussels”})

(v1)-[e1 :worksIn {start=”2020-01-01”}]->(v2)
(v1)-[e2 :livesIn {start=”2015-01-01”}]->(v3)
```

This graph contains three vertices `v1`, `v2` and `v3`, and two edges `e1` and `e2`. The vertex `v1` is associated with the content `{ Person=lbl, name="Jan Hidders" }`. The edge `e1` is associated with content `{ worksIn=lbl, start="2020-01-01" }`. Moreover, `e1` has tail vertex `v1` and head vertex `v2`.

End examples

Note: The described approach of viewing labels and properties as instances of the more general notion of attribute has an important disadvantage: it does not allow the same attribute name to appear as both a label and an attribute name on the same element. It is possible to adapt the formal definitions so that this is possible. One approach that has been suggested but has not been fully discussed in the working group is the following:

- Introduced the notion of *marked attribute name*, which is an attribute name that is marked to indicate that it represents a label. An attribute name, say *a*, that is marked is denoted as *:a*. We assume that any marked attribute is distinct from any attribute name and so attribute names do not start with ":".
- Define a set of *content keys*, which are either attribute names or marked attribute names.
- In the definition of property graph, redefine the notion of content so (1) it maps content keys to attribute values, rather than attribute names and (2) requires that a content key is mapped to `lbl` iff the content key is a marked attribute name.
- The denotation of a content record might either stay `{Person=lbl, name="Jan Hidders"}` or become `{:Person=lbl, name="Jan Hidders"}`.

Attribute types

We will assume that there is a predefined set of **attribute types**. This can include:

- **Atomic types** such as `STRING`, `INTEGER` and `DATE`
- The special type `LABEL` that indicates and attribute represents a label
- **Collection types** such `STRING ARRAY[10]` (an array of strings with maximum length 10, and `STRING MULTISET` (a multiset containing strings)
- **Record types** such as `{ Person: LABEL, name: STRING }`.
- **Union types** such as `STRING | INTEGER` describing a value that is of type `STRING` or of type `INTEGER`.
- Special generic types such as `anyPropertyValue` and `anyAttributeValue`.

We will assume that there is a subtype relationship defined over these types, so that for example `{ Person: LABEL, name: STRING }` is a subtype of `{ Person: LABEL }`. Next to this we assume that there are two types of semantics defined for each type: a **strict semantics** and a **weak semantics**. The strict semantics represents the attribute values that **strictly conform** to the type, where the weak semantics represents the attribute values that just **conform** to the type, by which we mean that it strictly conforms to the type or to a subtype of the type. For example, the record `{ Person=lbl, name="Jan Hidders" }` strictly conforms to the

type `{ Person: LABEL, name: STRING }` but only (weakly) conforms to `{ Person: LABEL }`. For both types of semantics we will assume that `LABEL` has only one conforming attribute value, which is the special value `lbl`.

Finally we will also assume that there is a notion of **type intersection** for all the attribute types, which we will denote as for example `{Person: LABEL, name: STRING} & {name: PERSONALNAME, address: STRING}`. Which would be equivalent to `{Person: LABEL, name: (STRING & PERSONALNAME), address: STRING}`, and for which we assume that it determines the greatest common subtype. We generalise this notion of intersection type for sets of attribute types, where the intersection over an empty set of types is assumed to denote the `EMPTY` type, i.e., the attribute to which no attribute value conforms.

Since the attribute types that are record types are precisely what is needed to describe the content associated with an element, we will also refer to them as **content types**. So in this document the terms record type and content type will be used as if they are synonymous.

This is all we will assume here for the attribute types. For a more in-depth discussion and an analysis of the assumptions in [GQL:EWD] we refer the reader to the companion report [LDBC PGS-B:BAS-02].

The syntax of graph-type cores

We start with a very simple type of graph-type core that only considers simple vertex types and edge types and does not consider any form of inheritance or subtyping. As mentioned in the introduction, we focus in this document on the graph-type core, i.e., the part of the schema that specifies the allowed types, and so leave aside schema constraints and graph-level attributes.

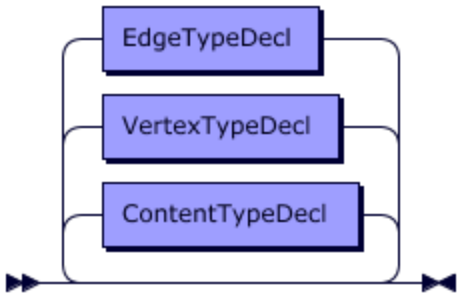
A proposal for a concrete syntax

The following is a provisional concrete syntax to represent graph-type cores. It is not necessarily meant as advice on how such a syntax should look, although it is considered a good example by most members of GS-Basic. An important feature is the usage of type variables that explicitly are marked as such by letting them start with some special symbol like “$”. This is mainly introduced so that in future extensions of the language it will always be clear in type definitions if a certain identifier represents a label or a type variable.

A grammar for the concrete syntax

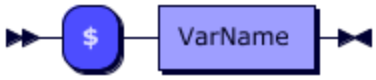
A graph-type core consists of a list of type specifications, which can be either vertex type specifications or edge type specifications. A graph-type core specification has the following syntax:

SchemaCore ::= (ContentTypeDecl | VertexTypeDecl | EdgeTypeDecl)*

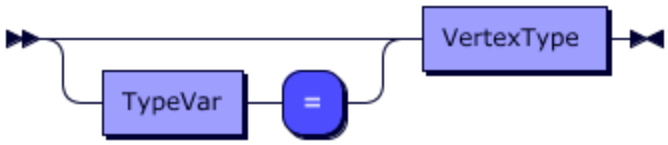


ContentTypeDecl ::= TypeVar "=" ContentType

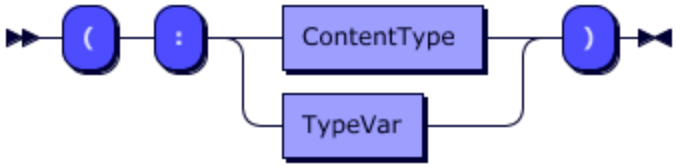
TypeVar ::= "$" VarName



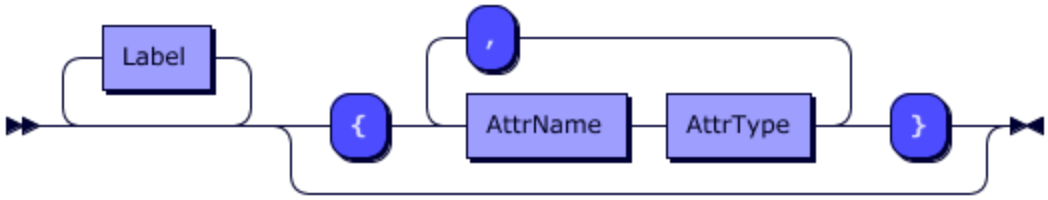
VertexTypeDecl ::= (TypeVar "=")? VertexType



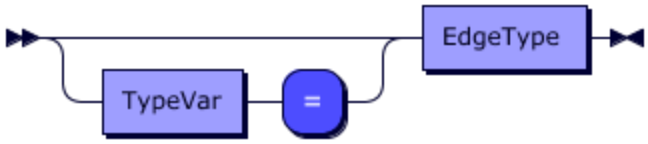
VertexType ::= "(" ":" (ContentType | TypeVar) ")"



ContentType ::= Label* ("{" AttrName AttrType ("," AttrName AttrType)* "}")?



EdgeTypeDecl ::= (TypeVar "=")? EdgeType



EdgeType ::= VertexType "-[" ":" ContentType "]->" VertexType

An example of a graph-type core is:

- `$personContent = :Person { name STRING, birthdate DATE }`
- `$person = (: $personContent )`
- `$city = (:City Place { name STRING, url URL })`
- `$country = (:Country Place { name STRING, url URL })`
- `$continent = (:Continent Place { name STRING, url URL })`
- `$livesIn = (:$person)-[:livesIn { start DATE }]->(:$city)`
- `$worksIn = (:$person)-[:worksIn { start DATE }]->(:$city)`
- `$cityLiesIn = (:$city)-[:liesIn]->(:$country)`
- `$countryLiesOn = (:$country)-[:liesOn]->(:$continent)`

The type names for content types (e.g., `$personContent`) and element types (e.g., `$city` and `$worksIn`) should be understood as a macro mechanism. So the above graph-type core is equivalent to:

- `(:Person { name STRING, birthdate DATE })`
- `(:City Place { name STRING, url URL })`
- `(:Country Place { name STRING, url URL })`
- `(:Continent Place { name STRING, url URL })`
- `(:Person { name STRING, birthdate DATE })`
  `-[:livesIn { start DATE}]->`
  `(:City Place { name STRING, url URL })`
- `(:Person { name STRING, birthdate DATE })`
  `-[:worksIn { start DATE}]->`
  `(:City Place { name STRING, url URL })`
- `(:City Place { name STRING, url URL })`
  `-[:liesIn]->`
  `(:Country Place { name STRING, url URL })`
- `(:Country Place { name STRING, url URL })`
  `-[:liesOn]->`
  `(:Continent Place { name STRING, url URL })`

Note: The choice of `$` as the marker for type variables might require more thought, as the symbol is already used for other purposes in SQL. We do however recommend that some symbol is chosen to distinguish variable names from labels. **End note**

Representing labels as attributes

Since labels are regarded as a special type of attribute, their denotation can be understood as a short-hand for denoting attributes of type `LABEL`. So, for example the content type

```
City Place { name STRING, url URL }
```

could also have been written as

```
{ City LABEL, Place LABEL, name STRING, url URL }
```

Note: Under the approach that was suggested earlier where we distinguish marked and unmarked attribute names to indicate properties and labels respectively, the alternative representation for the content type would be:

```
{:City LABEL, :Place LABEL, name STRING, url URL}
```

End note

We can allow using `&` between labels. So the content type `City Place { name STRING, url URL }` can also be written as `City & Place { name STRING, url URL }`. We expect this to be consistent with a later generalization where `&` is defined to be the type intersection operator. So the content type might then for example also be written as `City & Place & { name STRING } & { url URL }`.

Normalization of the concrete syntax

To simplify the task of defining the semantics of a graph-type core specification we will assume that it has been normalised so that (1) all labels are represented as attributes with type `LABEL` and (2) all variable substitutions have been made and (3) the result is a set of element types without duplicates. There is some variation possible in how this happens exactly. For example, there are different reasonable choices for the scope of variables and what happens with types that contain variables that are undefined. We consider this of minor importance, and so will in the remainder of this document simply assume that the specification is normalised and describes a set of element types. For an example of how a full normalization procedure could look like, the reader is referred to the companion report with formal definitions [LDBC PGS-B:BAS-02].

Abstract syntax

The formal and more abstract definition of a graph-type core that we will use for the definition of semantics as follows.

Definition: A **graph-type core** is defined as a set of element types where an element **type** is defined as either a **vertex type**, which consists of just a simple content type, or a **edge type**, which is assumed to consist of (1) a vertex type describing the tail type, (2) a simple content type describing the edge content and (3) a vertex type describing the head type. In each case we refer to the content type in the element type simply as **the content type** of the element type.

The semantics of graph-type cores

In this section we will present the proposals for the semantics of graph-type cores. We will start with the semantics that is considered preferable by most members of GS-Basic and PGSWG. This is followed by the alternatives that were also considered and even suggested by some members.

The at-least-one-match semantics

The semantics of a graph-type core is defined by defining when a certain property graph strictly conforms to a certain graph-type core, and when it just conforms. This is analogous to these notions for attribute values and their intention is the same: a property graph strictly conforms if it exactly matches the graph-type core, and it just (weakly) conforms if it in some sense can be regarded as an instance of a subtype of the graph-type core.

This is in turn based on the notion of **matching**, that defines when a certain element in a graph matches a certain type, which roughly means that all attributes required by the type are present in the element and contain a value of the correct type. We will distinguish two kinds of matching: *exact matching*, which requires that the type describes all attributes of the element, and *over matching*, which does not require this (and so allows the element to have more attributes than are specified by the type) . We start with defining these notions for content types:

> **Definition:** Given a content type and a record we say that the record **over matches** the content type if for every attribute in the content type it holds that the record has that attribute with a value that conforms to the attribute type specified in the content type. We say that the record is an **exact match** of the content type if it holds in addition that for every attribute in the record there is a corresponding attribute in the content type such that that attribute value strictly conforms to the attribute type.

For example, consider the content type `{ street: string, city: string }` then

- the record `{ street="Malet st", city="London" }` is an over match and also an exact match because the attributes *street* and *city* are defined and only those attributes and nothing else.
- the record `{ street="Malet st", city="London", country="UK" }` is an over match because the attributes *street* and *city* are defined but not an exact match because it has the attribute *country* which is not defined in the content type.
- the record `{ street="Malet st" }` is neither an over match nor an exact match because it is missing the attribute *city*, and

- the record `{ street=5, city="London", country="UK" }` is neither an over match nor an exact match because the value for the attribute *street* is a value that does not match the string type.

Based on the preceding notions we can generalise the notions of matching to the level of elements:

Definition: Given a property graph we say that an element in the graph is an **over match** of (or **conforms to**) an element type if the following holds:

- the content of the element is an **over match** of the content type in the type, and
- if the element is an edge then the content of the tail (head) vertex **over matches** the tail (head) type of the edge type.

We say that an element in the graph is an **exact match** of (or **strictly conforms to**) an element type if the following holds:

- the content of the element is an **exact match** of the content type in the type, and
- if the element is an edge then the content of the tail (head) vertex **exactly matches** the tail (head) type of the edge type.

Example

Consider the following graph-type core:

- `(:Person { name STRING, birthdate DATE })`
- `(:City { name STRING, url URL })`
- ```
  (:Person { name STRING, birthdate DATE })
    -[:worksIn { start DATE}]->
  (:City { name STRING, url URL })
  ```

Let's go through the following vertices and edges:

- `(: {Person=lbl, name="Jan Hidders"})`
  - **over match:** No
  - **exact match:** No
- `(: {Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  - **over match**: Yes
  - **exact match**: Yes
- `(: {Person=lbl, name="Jan Hidders", birthdate="1980-01-01", birthplace="Deventer"})`
  - **over match**: Yes
  - **exact match**: No
  ```

- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  `(v2 :{City=lbl, name="London", url="www.london.org"})`
  `(v1) -[e1 :worksIn]-> (v2)`
  - **over match (of edge** e1**)**: No
  - **exact match (of edge** e1**)**: No
- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  `(v2 :({City=lbl, name="London", url="www.london.org"})`
  `(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)`
  - **over match (of edge** e1**)**: Yes
  - **exact match (of edge** e1**)**: Yes
- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  `(v2 :{City=lbl, name="London", url="www.london.org"})`
  `(v1) -[e1 :worksIn {start="2020-01-01", foo= "bar"}]-> (v2)`
  - **over match (of edge** e1**)**: Yes
  - **exact match (of edge** e1**)**: No

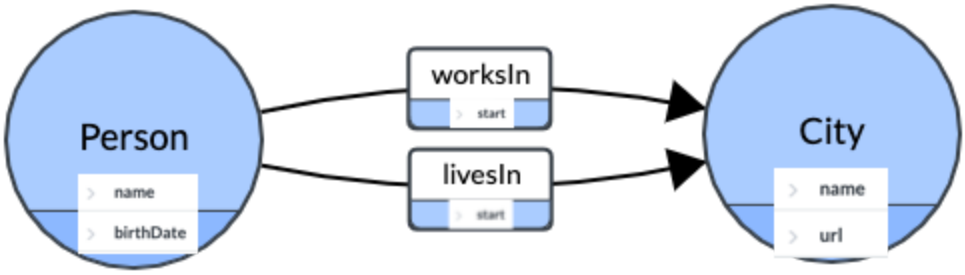
End example

Finally we define when a property graph conforms to a graph-type core.

Definition: We say that a property graph **conforms to** a graph-type core if for every element in the graph there is an element type in the core that it matches. We say that a property graph **strictly conforms to** a graph-type core if for every element in the graph there is an element type in the core that it exactly matches.

Example

Consider the following graph-type core:



Given the following Property Graph G1:

```
(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})
(v2 :{City=lbl, name="London", url="www.london.org"})
(v3 :{City=lbl, name="Brussels", url="www.brussels.org"})
```

```
(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)
(v1) -[e2 :livesIn {start="2015-01-01"}]-> (v3)
```

Property Graph G1 strictly conforms to the graph-type core because every element is an exact match

Given the following Property Graph G2:

```
(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01",
birthplace="Deventer"})
(v2 :{City=lbl, name="London", url="www.london.org"})
(v3 :{City=lbl, name="Brussels", url="www.brussels.org"})
(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)
(v1) -[e2 :livesIn {start="2015-01-01"}]-> (v3)
```

Property Graph G2 conforms to the graph-type core because the vertex v1 is an over match.

Given the following Property Graph G3:

```
(v1 :{Person=lbl, name="Jan Hidders"})
(v2 :{City=lbl, name="London", url="www.london.org"})
(v3 :{City=lbl, name="Brussels", url="www.brussels.org"})
(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)
(v1) -[e2 :livesIn {start="2015-01-01"}]-> (v3)
```

Property Graph G3 does NOT conform (hence does not strictly conform) to the graph-type core because the vertex v1 is not an over match.

End example

The combinatorial semantics

The semantics that we present here is strictly speaking not meant as an alternative proposal. We discuss it here because it played a role in earlier discussions of PGSWG, where it was referred to as the type-1 semantics (as presented in *"GS Basic, status update 23 March 2020"* [Basic:23-03-2020]). It is regarded by some GS-Basic members as too complicated, although some consider it as more in line with the usual semantics of conceptual data models, especially those that allow overlapping inheritance such as EER diagrams, UML class diagrams and ORM diagrams. However, it is widely agreed that it is probably not compatible with the leading view in WG3, and therefore is not given as a recommendation. Nevertheless, since it received quite a number of positive votes in a recent vote in PGSWG, we feel it is worth presenting here and comparing it to the other proposals.

Informally it can be described as allowing elements if their contents exactly match intersections of content types in the graph-type core. Equivalently this can be formulated as saying that each attribute in their content must be justified by at least one element type that they conform to.

Definition: We say that a property graph **combinatorially conforms to** a graph-type core if for every element in the graph it holds that (1) it conforms to at least one type in the graph-type core and (2) for every attribute of the element there is at least one element type in the graph-type core that the element conforms to and that justifies this attribute, i.e., the element type specifies an attribute with the attribute's name and with an attribute type that the attribute's value strictly conforms to.

Note that the preceding definition is equivalent to requiring that for each element it holds that the content of that element **strictly conforms** to the intersection type over the content types of the element types in the graph-type core that the element **conforms to**

Example

Consider the following graph-type core:

```
(:Person { name STRING, birthdate DATE })
(:Prof { status STRING, university STRING })
```

And consider the following Property Graph with only one vertex:

```
(v1 :{Person=lbl, Prof=lbl, name="Jan Hidders", birthdate="1980-01-01",
status="Lecturer", university="Birkbeck"})
```

- **Combinatorial Conform:** Yes because the vertex has all the content types for `Person` and `Prof` hence it is an over match. Additionally, it only has the content types of both `Person` and `Prof`, therefore it is also an exact match
- **Person conform:** Yes because it has the attributes for `Person` (`name` and `birthdate`) and others (status and university)
- **Prof conform:** Yes because it has the attributes for `Prof` (`status` and `university`) and others (`name` and `birthdate`)
- **Person strictly conform:** No because it has more attributes than defined for `Person`
- **Prof strictly conform:** No because it has more attributes than defined for `Prof`

Now consider the following Property Graph with only one vertex:

```
(v1 :{Person=lbl, Prof=lbl, name="Jan Hidders", birthdate="1980-01-01",
status="Lecturer", university="Birkbeck", foo="bar"})
```

- **Combinatorial Conform:** No because it has an attribute that is not defined in any of the types of which it is an over match: foo.
- **Person conform:** Yes because it has the attributes for `Person` (`name` and `birthdate`)

and others (status and university and foo)
- **Prof conform:** Yes because it has the attributes for `Prof` (`status` and `university`) and others (`name` and `birthdate` and `foo`)
- **Person strictly conform:** No because it has more attributes than defined for `Person`
- **Prof strictly conform:** No because it has more attributes than defined for `Prof`

End example

The combinatorial semantics essentially say that the content type of an element should exactly match the intersection of the content types of all element types which the element over-matches. So, for edge types the head and tail types are ignored for the exact match. To understand why, consider the following example.

```
$message = (:Message { creationDate DATE, content STRING })
$comment = (:Comment { public BOOL })


(:$comment)-[:isReplyTo]->(:$message)
```

The intention of the combinatorial conformance is to interpret the semantics of types based on over matching rather than exact matching and to allow vertices to fully match with combinations of vertex types. In this case the type `$comment` can for example be combined with the type `$message`. Under this interpretation it makes sense to let the schema require for `isReplyTo` edges that the tail also over-matches `$comment`. Therefore we allow that such edges end in vertices that fully match a combination of `$message` and `$comment`. Observe that if in the combinatorial semantics we would have required for edges that they fully match the combination of a subset of the edge types, this would not have been allowed.

For a more in-depth analysis of the relationship between normal conformance and combinatorial conformance, the reader is referred to the companion report [LDBC PGS-B:BAS-02], which gives a fully formal definition and formally compares the two.

A note on computational complexity: The computational complexity of schema validation under combinatorial semantics may seem exponential at first sight since it quantifies over subsets of the graph-type core. However, it is sufficient to check the desired property for the set that consists of all element types that the element conforms to, and this can be checked in polynomial time.

The exactly-one-match semantics

One might consider making the definition of conformance more simple and strict and require that for each element there is exactly one fully matching type in the graph-type core. This suggestion was made and argued for in [LDBC PGS:AG-05]. The formal definition for conformance would in that case be changed to the following (with changes highlighted in grey).

Definition: We say that a property graph **conforms to** a graph-type core if for every element in the graph there is exactly one element type in the graph-type core such that the element **conforms** to the element type. We say that the graph **strictly conforms to** the graph-type core if for every element in the graph there is exactly one element type in the graph-type core that the element **strictly conforms** to.

It should be noted that this might change the semantics and lead to unexpected exclusion constraints for certain types of semantics of attribute types. To illustrate this, consider the following graph-type core;

- `$personWithLongName = (:Person { name VARCHAR(50) })`
- `$personWithShortName = (:Person { name VARCHAR(15) })`

Under the at-least-one-match semantics this has the following strictly conforming graph:.

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Person { name="George Walsh" }`

However, under exactly-one-match semantics this graph no longer strictly conforms, since `v2` matches both `$personWithLongName` and `$personWithShortName`. Similar behaviour occurs with other property value types that can overlap, like record types with optional attributes, or union types. These cannot be removed by rewriting if they are nested inside collection types.

On the other hand, if it holds that all attribute types are disjoint, i.e, for all two distinct attribute types it holds that there is no attribute value that strictly conforms to both types, then the exactly-one-match semantics and the at-least-one-match semantics (i.e., the first semantics that was presented) are equivalent for strict conformance.

This issue also manifests itself for weak conformance. Consider the following graph-type core:

- ```
  $personWithAddress = (:Person Local { address { street STRING,
                                                  number STRING,
                                                  city STRING })
  ```
- ```
  $personWithIntlAddress = (:Person { address { street STRING,
                                                number STRING,
                                                city STRING,
                                                country STRING })
  ```

Under weak conformance as defined by the exactly-one-semantics the following graph would not weakly conform.

- `(v1 :Person Local { street="Atomiumplein", number="1", city="Brussels", country="Belgium" })`

This is because it weakly conforms to both types. Also here we can observe that the at-least-one-match semantics and the exactly-one-match semantic are equivalent if we assume that there is no attribute value that weakly conforms to two distinct attribute types. However, this assumption does not hold if we allow subtyping for record types.

The isolation-aware semantics

The at-least-one-match semantics has an implicit assumption that the content types in an edge type in a graph-type core that describe the head and tail vertices correspond to the content type of a vertex type in the graph-type core. We will call the content types for which this is **not** the case **dependent content types**, and the vertex types they define **dependent vertex types**. If a graph-type core has dependent content types then under the current semantics the edge types in which they appear cannot be populated. For example, consider the following graph-type core declaration:

- `$PersonContent = Person { name STRING, birthdate DATE }`
- `$worksInContent = worksIn { start DATE}`
- `$CityContent = City { name STRING, url URL }`
- `(: $PersonContent )-[: $worksInContent ]-> (: CityContent )`

Note that in this declaration there are no vertex type declarations, and after substitution we obtain the following graph-type core:

- ```
  (:Person { name STRING, birthdate DATE })
    -[:worksIn { start DATE}]->
  (:City { name STRING, url URL })
  ```

In this case only the empty graph conforms, since the graph-type core contains no vertex types and so no graph containing vertices can conform.

One way to solve this is to allow vertices to exist already if they participate in an edge that is justified to exist, i.e., matches one of the edge types. We illustrate here how to adapt the definition for strict conformance, but the same principles can be applied to conformance and combinatorial conformance.

We adapt the definition of strict conformance as follows: (changed parts marked in grey):

Definition: We say that a property graph **strictly conforms to** a graph-type core if

- for every edge in the graph there is a edge type in the graph-type core which it strictly
  ```

conforms to and
- for every isolated vertex in the graph there is a vertex type in the graph-type core that it strictly conforms to.

Under these semantics the following graph strictly conforms to the preceding graph-type core.

- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
- `(v2 :{City=lbl, name="London", url="www.london.org"})`
- `(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)`

The next graph however does not strictly conform because it contains an isolated vertex without having a matching independent type in the graph-type core.

- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`

This change in semantics has advantages and disadvantages:
- **Good:** It adds expressive power and gives a meaningful semantics for graph-type cores that otherwise would only allow the empty graph.
- **Good:** It is similar to existing notions in conceptual data models such as ORM, where object types have to be declared explicitly as *independent* if their instances should be able to exist without participating in any non-identifying relationships.
- **Bad**: It makes the semantics a bit more complex and implies sophisticated database constraints that might not be obvious.
- **Bad**: In the case of conformance it might make certain forms of factorisation harder.

To illustrate the point about sophisticated database constraints, consider the following graph-type core:

- ```
  (:Person { name STRING, birthdate DATE })
    -[:worksIn { start DATE}]->
  (:City { name STRING, url URL })
  ```
- ```
  (:Person { name STRING, birthdate DATE })
    -[:livesIn { start DATE}]->
  (:City { name STRING, url URL })
  ```

Under the adapted semantics this implies the following two constraints (formulated in the syntax that is close the one that is discussed in the working group for keys and cardinality constraints):

```
WHERE (p :Person) REQUIRE
  (THERE IS AT LEAST ONE wi SUCH THAT (p)-[wi :worksIn]->()) OR
  (THERE IS AT LEAST ONE li SUCH THAT (p)-[li :livesIn]-())

WHERE (c :City) REQUIRE
  (THERE IS AT LEAST ONE wi SUCH THAT ()-[wi :worksIn]->(c)) OR
```

```
(THERE IS AT LEAST ONE li SUCH THAT ()-[li :livesIn]-(c))
```

For the point about factorisation consider the following graph-type core:

- `(:person manager)`
- `(:person employee)`
- `(:person)-[:knows]->(:person)`

This would now allow vertices with just the label `person`, and not those with `manager` or `employee`, which might not be the intended meaning. We could try to remedy this by indicating that we allow subtypes, by for example replacing `:` with `<:`. So the edge type could then be:

- `(<:person)-[:knows]->(<:person)`

However, that would then allow too much, since we could have a vertex with any set of labels in such an edge, as long as the set of labels contains the label `person`.

The homomorphism-based semantics

Another alternative semantics that was proposed is based on homomorphisms between property graphs and graph-type cores, and was inspired by [Bonifati:2019]. It is based on the observation that graph-type cores can be interpreted as graphs, where the vertex types are vertices and edge types are edges that connect these vertices (assuming that the vertex types in the edge type are also independently present in the graph-type core). As a consequence there is a natural notion of homomorphism from property graphs to graph-type cores. In the GS-Basic subgrup a case for this approach was made in [LDBC PGS:AG-10r1].

For the sake of this presentation we will discuss this in the context of strict conformance semantics, but the same principles can also be applied to conformance and combinatory conformance semantics. We start with the notion of homomorphism that we will base the semantics on.

Definition: Given a property graph and a graph-type core we define a **homomorphism** from the graph to the graph-type core as a function that maps the elements of the graph to an element type in the graph-type core such that:

1. every vertex in the graph is mapped to a vertex type it strictly conforms to and
2. every edge in the graph is mapped to an edge type such that (1) the tail (head) vertex of the edge is mapped to the tail (head) type of the edge type and (2) the content of the edge strictly conforms to the content type of the edge type.

We can then redefine strict conformance such that a property graph conforms to a graph-type core if there is a homomorphism from the property graph to the graph-type core.

To illustrate these semantics, consider the following graph-type core;

- `$personWithLongName = (:Person { name VARCHAR(50) })`
- `$personWithShortName = (:Person { name VARCHAR(15) })`
- `$degree = (:Degee { level VARCHAR(3), level VARCHAR(25)})`
- `$hobby = (:Hobby { description VARCHAR(100) })`
- `($personWithLongName-[:hasDegree]->(:$degree)`
- `($personWithShortName-[:hasHobby]->(:$hobby)`

Under the homomorphism-based semantics the following graph strictly conform the the preceding schema.

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Person { name="George Walsh" }`
- `(v3 :Degree { level="MSc" topic="Philosophy"}`
- `(v4 :Hobby { description="Bird watching" }`
- `(v1) -[:hasDegree]-> (v3)`
- `(v2) -[:hasHobby]-> (v4)`

The following graph does not strictly confirm under homomorphism-based semantics:

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Degree { level="MSc" topic="Philosophy"}`
- `(v3 :Hobby { description="Bird watching" }`
- `(v1) -[:hasDegree]-> (v2)`
- `(v1) -[:hasHobby]-> (v3)`

It is possible to combine this approach with the ideas for dealing with dependent vertex types that were discussed in the previous section. This again allows us to give semantics to vertex types that only appear in edge types and not independently. For this we need a slightly more sophisticated notion of homomorphism.

Definition: Given a property graph and a graph-type core we define an **isolation-aware homomorphism** from the graph to the graph-type core as a function that maps the elements of the graph to an element type in the graph-type core or to a vertex type that is the head or tail type of an edge type in the graph-type core, such that:

1. every isolated vertex in the graph is mapped to a vertex type in the graph-type core,
2. every vertex in the graph is mapped to a vertex type it strictly conforms to and
3. every edge in the graph is mapped to an edge type such that (1) the tail (head) vertex of the edge is mapped to the tail (head) type of the edge type and (2) the content of

> the edge strictly conforms to the content type of the edge type.

Subsequently we can define strict conformance such that a property graph strictly conforms to a graph-type core if there is an isolation-aware homomorphism from the property graph to the graph-type core.

To illustrate the difference with the previous semantics that is not isolation-aware, consider the following adapted graph-type core:

- `$personWithLongName = :Person { name VARCHAR(50) }`
- `$personWithShortName = :Person { name VARCHAR(15) }`
- `$degree = (:Degee { level VARCHAR(3), level VARCHAR(25)})`
- `$hobby = (:Hobby { description VARCHAR(100) })`
- `($personWithLongName-[:hasDegree]->(:$degree)`
- `($personWithShortName-[:hasHobby]->(:$hobby)`

Note that `$personWithLongName` and `$personWithShortName` now define content types rather than vertex types. The following graph strictly conforms under isolation-aware homomorphism-based semantics:

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Person { name="George Walsh" }`
- `(v3 :Degree { level="MSc" topic="Philosophy"}`
- `(v4 :Hobby { description="Bird watching" }`
- `(v1) -[:hasDegree]-> (v3)`
- `(v2) -[:hasHobby]-> (v4)`

However, the following graph does not, since `v2` is now an isolated vertex for which there is no matching independent vertex type:

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Person { name="George Walsh" }`
- `(v3 :Degree { level="MSc" topic="Philosophy"}`
- `(v1) -[:hasDegree]-> (v3)`

These changes in semantics have advantages and disadvantages:

- **Good:** The isolation-aware semantics gives a meaningful semantics for graph-type cores with vertex types that only occur in edge types.
- **Good:** The homomorphism semantics adds expressive power, although not as much as when the schema would actually be a graph giving abstract identity to vertex types.
- **Bad**: The added expressive power is quite subtle, and probably hard to understand by most users.

- **Bad**: Both have the disadvantage that, if distinct types can overlap, the computational complexity of validation becomes **intractable**, since it becomes similar to the general problem of checking for the existence of graph homomorphisms.

Summary

Main results and contributions

The presented contribution in this report consists of a proposal for the syntax for graph-type cores and several proposals for their semantics. For each proposal for the semantics a motivation is given for why it can be seen as an improvement over the at-least-one-match semantics, and the trade-offs are discussed.

We feel that each proposal offers an interesting insight, even if it is not considered the preferred one, since it highlights a weak point of the other proposals. For example the combinatorial semantics highlights the need to allow for certain sets of types to be combined. The isolation-aware semantics highlights that there is an implicit assumption that vertex types that are part of edge types also appear as independent vertex types in a graph-type core. The alternative proposals therefore provide insight into possible future extensions of the preferred semantics, to compensate for these weak points.

Observations

The main observations by the GS-Basic group are as follows:

- The main underlying ideas of the semantics seem well-established, although there is no consensus on which proposal is preferred. However, the at-least-one-match semantics proposal in this document seems to have the most support.
- The semantics of graph-type cores is for simple schemas similar to that of conceptual data models, but there is a clear difference when it comes to allowing overlap of types. Here conceptual data models tend to have a notion of overlapping inheritance which explicitly allows objects to belong to combinations of types. Under the current semantics such combinations would have to be explicitly added to the graph-type core. It will be an interesting challenge to add a similar construct to property graph schemas while preserving the computational complexity of validation.
- The notion of (weak) conformance seems to follow naturally and the subtype relationship it implies for graph-type cores seems to satisfy the expectations for such a relationship. To illustrate this, consider a query that retrieves vertices by matching a graph pattern, from a graph that is expected to strictly conform to a certain graph-type core. We can then derive the expected vertex type for the retrieved vertices. If the same query is executed over a graph that only weakly conforms, then resulting vertices will still conform to the derived vertex type.

- The at-least-one-match semantics proposal seems straightforwardly extensible, as is for example illustrated by the extension with optional attributes, as is shown in the companion report [LDBC PGS-B:BAS-02.
- The discussion concerning the exactly-one-match semantics (see Section The exactly-one-match semantics) illustrates that assumptions concerning the type system for attribute values, especially those concerning overlap of types, can have repercussions for the semantics at schema level. It is therefore important to consider these as related issues.

Open Questions

There are several open questions and issues that we intend to address in future work. We list them here.

What is an appropriate syntax for type variables?

In the provisional concrete syntax used in this document type variables are made distinct from labels by prefixing them with “$”. It may seem a detail, but picking an appropriate syntax that distinguishes these variables form any other type of variable or parameter will be an important choice.

What is the syntax and semantics for inheritance?

Although this topic was intentionally avoided in this report, there have already been suggestions on how to describe inheritance, such as in [LDBC PGS:AG-05]. The basic idea is to allow type variables in locations where in the current syntax labels are expected. A graph-type core specification might look as follows:

- `$message = (:Message { creationDate DATE, content STRING })`
- `$post = (:Post $message { language STRING })`
- `$comment = (:Comment $message)`
- `(:$comment)-[:isReplyTo]->(:$message)`

The inclusion of a type variable, like `$message` in `$post` signifies that the latter defines a subtype of the first. This subtype would be computing by taking the intersection type specification without the variable (`(:Post { language STRING })`) and the type associated with the variable (`(:Message { creationDate DATE, content STRING })`).

An important issue for this type of semantics is that under strict conformation the above graph-type core would not allow `isReplyTo` edges to end in a vertex of type `post` or `comment`, but only in vertices of type `$message`, which seems intuitive given the usual interpretation of inheritance.

Related to the issue of inheritance is the issue of abstract types. It might be for example that in the above graph-type core we would like to indicate that the type `$message` is abstract and only serves as a supporting type to define the two subtypes `$post` and `$message`. This can already be achieved by declaring the type `$message` as a content type, but that would allow this content type accidentally benign used as a content type for an edge type. Moreover, this cannot be used for abstract edge types. So an explicit keyword `ABSTRACT` that would modify a type to be abstract would probably be a useful construct to have.

How to introduce open types?

Given the clear need for introducing features in the schema language that allow it to be partially more open in certain places, even under strict semantics, raises the question of what features and constructs would be appropriate.

At the attribute level there are obvious candidates such as explicit union types and the generic types `anyPropertyValue` and `anyAttributeValue`. These would already allow to create schemas with a certain openness under strict semantics.

At the level of element types an obvious candidate (suggested in [LDBC PGS:AG-05]) is to mark element types explicitly to indicate that they allow instances of subtypes, even under strict conformance. For example, if we indicate this by replacing the ":" with "<:", then the previous core might be changed into:

- `$message = (:Message { creationDate DATE, content STRING })`
- `$post = (:Post $message { language STRING })`
- `$comment = (:Comment $message)`
- `(:$comment)-[:isReplyTo]->(<:$message)`

Under this graph-type core an `isReplyTo` edge would, under strict semantics, be allowed to end in a vertex which is of a subtype of `$message`, so for example of type `$post` or type `$comment`. Note that this would still keep the graph-type core closed in the sense that we cannot have vertices that do not strictly conform to one of the vertex types. This would change if we indicated for one of the vertex types that it allows subtypes:

- `$message = (:Message { creationDate DATE, content STRING })`
- `$post = (:Post $message { language STRING })`
- `$comment = (<:Comment $message)`
- `(:$comment)-[:isReplyTo]->(<:$message)`

This would allow vertices in a strictly conforming graph if they strictly conform to a subtype of `(:Comment $message {})`, or to be more explicit, a subtype of `(:Message Comment { creationDate DATE, content STRING })`. Note that this means they can have next to the explicitly specified attributes, additional attributes of any type.

While the additions described above are clearly useful, it is not clear if this covers most of the practical uses cases in an intuitive manner, and if additional constructs are needed for that.

Which constructs can approximate conceptual data models?

As discussed in Section Observations a feature that is hard to model in the given syntax is that which in Enhanced ER models is called overlapping inheritance with multiple subtypes. This would require some constructor that given a certain number of types generates all combinations of those types. One such construct is the power union type, as was suggested in [LDBC PGS:AG-08]. It could be used as follows:

- `$employee = (:Employee { … })`
- `$manager = (:Manager $employee { … })`
- `$engineer = (:Engineer $employee { … })`
- `$mentor = {:Mentor $employee { … })`
- `(: PUT($manager, $engineer, $mentor) )`

Here `PUT` would be a short-hand that specifies several element types, namely all those that are the intersection type over a **non-empty** subset of the set `{ $manager, $engineer, $mentor }`. So that would include for example `$manager & $engineer` and `$engineer & $mentor` and `$manager & $engineer & $mentor` . The above graph-type core would therefore be equivalent to:

- `$employee = (:Employee { … })`
- `$manager = (:Manager $employee { … })`
- `$engineer = (:Engineer $employee { … })`
- `$mentor = (:Mentor $employee { … })`
- `(: $manager )`
- `(: $engineer )`
- `(: $mentor )`
- `(: $manager & $engineer )`
- `(: $manager & $mentor )`
- `(: $engineer & $mentor )`
- `(: $manager & $engineer & $mentor )`

Further analysis of this construct is needed to see if it indeed can cover in an intuitive manner all cases with overlapping inheritance, and if it preserves the tractable computational complexity of validation.

What is the semantics of undirected-edge types?

An obvious extension of the syntax for dealing with undirected edges would be to indicate in the syntax of an edge type that it describes an undirected edge, e.g.:

- `(:$guest)-[:isFriendOf]-(:$member)`

However, if like in this case the tail and head type are different, that raises the question of what the semantics of this is. There seem to be several reasonable options here, such as:

1. This should not be allowed and has no semantics. Undirected edges are inherently symmetric, and so an asymmetric type does not make sense.
2. It means that both the incident vertices should be of type `$guest & $member`, so be of type `$guest` as well as of type `$member`. In other words, a guest can only be a friend of a member if they are themselves a member, and a member can only be a friend of a guest if that member is a guest. This is close to the idea that an undirected edge actually represents two edges, one in each direction.
3. It means that one incident vertex should be of type `$guest` and the other of type `$member`, or vice versa. That would allow a guest to be a friend of a member, even if that guest is not themselves a member. This is close to the idea that an undirected is like a directed edge except that it can be navigated in both directions.

We expect that more discussion will be needed on this issue before we can settle on what the preferred semantics is.

Why and how to introduce nominal typing?

In the report only structural typing has been discussed, which means that the type name does not play a role in the semantics of a type. Since SQL uses nominal typing it might be appropriate to incorporate that into edge types and node types. Usually this means that the instance of the type is annotated with the type name of the type it is declared to be an instance of. However, it is not yet clear if this is really needed, and if it would involve changes to the underlying data model.

Conclusion

The semantics of a graph-type core, which contains only a set of vertex and edge types, may seem a straightforward issue. However, as will be obvious from this document, it is a topic with quite a few unexpected and subtle complications. So looking back it is not surprising that the result of the discussions in GS-Basic have led to several proposals that all have their benefits. It is for that reason that we have decided to present all of them in full detail in this report, even though the at-least-one-match semantics has the widest support. We hope that the insights presented in this report and its companion report will help the ISO/IEC JTC1 SC32 WG3 working group in determining the syntax and semantics for GQL graph types.

LDBC PGS-B:BAS-02r1

GS-Basic Formal definitions report

Editors: Jan Hidders and Juan Sequada
Authors: Dominik Tomaszuk

Table of contents

References

[LDBC PGS-B:BAS-01] GS-Basic Overview report, Ed. Jan Hidders

[GQL:DATA MODEL] ISO/IEC JTC1 SC32 WG3 SXM-030r3 *"The GQL property graph data model "*

[GQL:EWD] ISO/IEC JTC1 SC32 WG3 WG3:MMX-010r2 *"GQL Early Working Draft V4.2"*

[LDBC PGS:JH-03r7] *On the syntax and semantics of GS-Basic without inheritance*, Ed. Jan Hidders and Juan Sequeda, https://docs.google.com/document/d/13ECQZgqJfuBhHlCyxQfkNFjwG2jzwv5Md16ns9VyZWw/edit?usp=sharing

[Hidders:1995] Hidders, J. (1995). Union-Types in Object-Oriented Schemas. In P. Atzeni & V. Tannen (Eds.), *Database Programming Languages (DBPL-5), Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Umbria, Italy, 6-8 September 1995* (p. 2). Springer. http://ewic.bcs.org/content/ConWebDoc/5191

[Basic:23-03-2020] GS-Basic: Progress report: 23 March 2020

[Bonifati:2019] Bonifati, A., Furniss, P., Green, A., Harmer, R., Oshurko, E., Voigt, H.: Schema validation and evolution for graph databases. In: Proceedings of the 38th International Conference on Conceptual Modeling (ER) (2019)

[Balsters:1991] Herman Balsters, Maarten M. Fokkinga, *Subtyping can have a simple semantics,* Theoretical Computer Science, Volume 87, Issue 1, 1991, Pages 81-96, ISSN 0304-3975, https://doi.org/10.1016/S0304-3975(06)80005-8.

Goal of this document

The goal of this document is to serve as a companion document to provide the full formal definitions of the concepts presented in [LDBC PGS-B:BAS-01]. To make this document readable as a standalone document, it replicates informal explanations and examples from that document. In addition it also contains the following additional content that was not presented in [LDBC PGS-B:BAS-01]:

- an example of a mapping from the concrete syntax to the abstract syntax (see Section Mapping the concrete syntax to the abstract syntax)
- a discussion of the mapping between the at-least-one-match semantics and the combinatorial semantics, (see Section Comparing normal conformance and combinatorial conformance)
- an extension of the at-least-one-match semantics for dealing with optional attributes (see Section graph-type cores with optional attributes) and
- an in-depth analysis of the assumptions concerning the type system that governs that attribute values (see Section An analysis of the assumptions in WG3:MMX-010r2).

Basic definitions

We start with recalling the main definitions that will be relevant for the discussion. For the definition of property graph we assume the existence of the following sets:

- A countably infinite set $\mathcal{A}$ of attribute names
  - this generalises over property names and labels.
- A countably infinite set $\mathcal{V}$ of attribute values
  - it includes a special value **lbl**, which is meant to be used as the value of an attribute to indicate that it represents a label and not a property

Examples

- $\mathcal{A}$ = {name, url, Person, City, ... }
- $\mathcal{V}$= {Jan, London, 2020-01-01, 1, 2 , …, **lbl**}

End examples

Property graphs

Definition: A property graph is defined as $G = (V, E, \varrho, \alpha)$ where

- V is a set of vertex identities,
- E is a set of edge identities such that $V \cap E = \varnothing$,
- $\varrho : E \to (V \times V)$ a total function mapping edge identities to pairs of vertex identities and
- $\alpha : (V \cup E) \to (\mathcal{A} \rightharpoonup \mathcal{V})$ a total function mapping edge/vertex identities to a finite partial function that maps attribute names to attribute values.

Examples

Consider the following Property Graph

```
(v1 :Person {name = "Jan Hidders"})
(v2 :City {name = "London"})
(v3 :City {name = "Brussels"})

(v1)--[e1 :worksIn {start="2020-01-01"}]-->(v2)
(v1)--[e2 :livesIn {start="2015-01-01"}]-->(v3)
```

This Property Graph is represented with the following notation from the formal definition:

$G = (V, E, \varrho, \alpha)$ where

- $V = \{v_1, v_2, v_3\}$
- $E = \{e_1, e_2\}$
- $\varrho(e_1) = (v_1, v_2)$
- $\varrho(e_2) = (v_1, v_3)$
- $\alpha(v_1) = \{\text{Person} \mapsto \textbf{lbl}, \text{name} \mapsto \textit{“Jan Hidders”}\}$
- $\alpha(v_2) = \{\text{City} \mapsto \textbf{lbl}, \text{name} \mapsto \textit{“London”}\}$
- $\alpha(v_3) = \{\text{City} \mapsto \textbf{lbl}, \text{name} \mapsto \textit{“Brussels”}\}$
- $\alpha(e_1) = \{\text{worksIn} \mapsto \textbf{lbl}, \text{start} \mapsto \textit{“2020-01-01”}\}$
- $\alpha(e_2) = \{\text{livesIn} \mapsto \textbf{lbl}, \text{start} \mapsto \textit{“2015-01-01”}\}$

End examples

Note: The described approach of viewing labels and properties as instances of the more general notion of attribute has an important disadvantage: it does not allow the same attribute name to appear as both a label and an attribute name on the same element. It is possible to adapt the formal definitions so that this is possible. One approach that has been suggested but has not been fully discussed in the working group is the following:

- Introduced the notion of *marked attribute name*, which is an attribute name that is marked to indicate that it represents a label. An attribute name, say a, that is marked is denoted as $:a$. We assume that any marked attribute is distinct from any attribute name and so does not start with “:”.
- Define a set of *content keys*, which are either attribute names or marked attribute names.
- In the definition of property graph, replace $\mathcal{A}$ with the set of content keys, and require that for any element x it holds that $\alpha(x)$ maps content keys to **lbl** iff they are marked attribute names.
- The denotation of a content record might either stay `{Person=lbl, name="Jan Hidders"}` or become `{:Person=lbl, name="Jan Hidders"}`.

End note.

Attribute types

We start with stating our assumptions concerning attribute values and attribute types. Most of these assumptions originate from [GQL:EWD], and we will discuss where and why we differ.

The attribute type lattice

We presume the existence of the following sets and relationships between them:

- A set $\mathcal{T}_{attr}$ of attribute types describing attribute values.

  - It includes a special attribute type **label** that will be used to indicate in a schema to indicate that a certain attribute represents a label.
  - We assume there is a subtype relation ≤ defined over $\mathcal{T}_{attr}$, which is a partial order that defines a lattice.
  - We assume in addition that ≤ defines a lattice, i.e., for any two attribute types $\tau, \sigma \in \mathcal{T}_{attr}$
    - there is a unique type that is the largest common subtype of τ and σ, which we will denote as $\tau \wedge \sigma$ and
    - there is a unique type that is the smallest common supertype of τ and σ, which we will denote as $\tau \vee \sigma$.
  - We assume a special attribute type **empty** that is the smallest element in the lattice.
- For each attribute type $\tau \in \mathcal{T}_{attr}$ there is a possibly empty set $[\![\tau]\!] \subseteq \mathcal{V}$ containing all attribute values that strictly conform to type τ. We will also refer to this set as the strict semantics of τ.
  - In particular $[\![$**label**$]\!]$ = { lbl } and $[\![$**empty**$]\!] = \varnothing$
  - We assume that for each attribute value $v \in \mathcal{V}$ there is an attribute type $\tau \in \mathcal{T}_{attr}$ such that $v \in [\![\tau]\!]$.

The notation for the largest common subtype is generalised for sets of attribute types, so that we can write for example $\tau_1 \wedge \ldots \wedge \tau_n$ for a finite set of attribute types $\{\tau_1, \ldots, \tau_n\}$. Given the stated assumptions of the type lattice this has a well-defined unique semantics for non-empty finite sets. If the set is empty, we will assume $\tau_1 \wedge \ldots \wedge \tau_n$ denotes the attribute type **empty**.

Examples

- $\mathcal{T}_{attr}$ = {**string**, **int**, **date**, **label**, **empty**, **anyPropVal**, **anyAttrVal**}
- $[\![$**string**$]\!]$ = {“Jan Hidders”, “London”, “Brussels”, … }
- $[\![$**int**$]\!]$ = {1, 2, … }
- $[\![$**date**$]\!]$ = {2020-01-01, 2015-01-01, … }
- **[[label]]** = {**lbl**} // This is fixed
- **[[empty]]** = $\varnothing$
- **[[anyPropVal]]** = $[\![$**string**$]\!] \cup [\![$**int**$]\!] \cup [\![$**date**$]\!]$
- **[[anyAttrVal]]** = **[[anyPropVal]]** ∪ {**lbl**}
- **Subtyping lattice:**
  - **empty** < **int**, **empty** < **date**, **empty** < **label**
  - **int** < **string**, **date** < **string**
  - **string** < **anyPropVal**
  - **anyPropVal** < **anyAttrVal**, **label** < **anyAttrVal**

End examples

On the meaning of subtyping

As might be clear from the example, there are very few restrictions on the type lattice, except that ≤ must be a partial order. Therefore it allows subtype relationships, such as **int < string** and **date < string,** which might or might not be intuitive within a certain setting. The main reason for this is that we are trying to define a framework that is as generic and extensible as possible and therefore has as few restrictions as possible. Another reason is that this framework is meant to serve as a vehicle to study what happens if certain common assumptions are added or removed. It is certainly the case that in some settings additional restrictions are desirable, either for implementation reasons or for conceptual simplicity or both.

The two kinds of subtyping

There is however an underlying philosophy that motivates why certain common restrictions that are found in other type systems are omitted. First of all, we distinguish two kinds of subtype-supertype relationships:

1. The supertype describes a larger set of values that is **less restricted** than the set of values described by the subtype. As an example consider a supertype that describes all strings and a subtype that describes a subset that matches a certain regular expression. In this case the semantics of the subtype is simply a subset of that of the supertype.
2. The supertype describes a set of values that are **abstractions** of values in the subtype. By abstractions we mean here that certain parts of the value are omitted. As an example consider a supertype that describes records with two fields and a subtype that describes records with an additional field.

Given these two kinds of subtype relationships it seems reasonable to allow a subtype relationship between any two types where all values in the potential subtype can be interpreted as values in the supertype, which might possibly involve abstraction, i.e., the omission of information. In other words, there should be an intuitive coercion function that maps the values of the subtype to values of the supertype. This function is required to be total, but is not required to be injective. One example of such a function is the projection of records on a particular set of fields. It is also possible that the coercion function is simply the identity function, as might be the case where the subtype describes a set of strings that is a sublanguage of the supertype. So we explicitly allow that the strict semantics of a subtype and a supertype overlap.

Consistent behaviour

To satisfy the **Liskov substitution principle** the collection of coercion functions must provide a consistent view on which values are conceptually the same and which values are abstractions of each other. For example, if there are multiple paths in the type lattice between a subtype and a supertype, then all these paths should have the same associated coercion function. Therefore additional constraints are required so that the coercion functions behave consistently. To understand what these constraints are, we must first define more precisely what we mean here

by consistent behaviour.

By consistent behaviour we mean that the type lattice with strict order $\tau < \sigma$ and associated family of coercion functions $K[\tau < \sigma] : [\![\tau]\!] \rightarrow [\![\sigma]\!]$ can be interpreted consistently. This means that we can interpret all elements in $\mathcal{U}$ as representing a conceptual value, like for example the integer number 5. This number might be represented by one attribute value in the type **int8** and by another attribute value in **int32**, but the interpretation would map each of these to the same conceptual number 5 to represent the fact that for users these actually represent the same thing. In addition these conceptual values would have an ordering that defines when one is considered to be an abstraction of another. For example, the record {a ↦ 5} could be a considered a more abstract version of the record {a ↦ 5, b ↦ **true**}. To be a valid interpretation it must satisfy two constraints.

1. The first constraint represents the intuition that the attribute values in the strict semantics of a certain type represent conceptual values at the same abstraction level. This means that if we take a certain conceptual value, like the record {a ↦ 5, b ↦ **true**}, there can be at most one attribute value that represents it at the abstraction level of the type {a ↦ **integer**}. Stated more precisely: if the interpretations of two attribute values in the strict semantics of an attribute type are the same conceptual value or abstract versions of the same conceptual value, then these attribute values must be identical.
2. The second constraint captures that the coercion functions are consistent with the notion of abstraction in the interpretation. This means that they map one attribute value to another exactly if the interpretation of the second is an equally or more abstract version of the interpretation of the first. So it should for example map the record {a ↦ 5, b ↦ **true**} to the record {a ↦ 5}, but not to the record {a ↦ 6}.

Formalisation of consistent behaviour

More formally, we can define a valid interpretation as consisting of:

1. A set $\mathcal{W}$ containing the conceptual values that are represented by the attribute values.
2. A partial order $\leq_{\mathcal{W}}$ over $\mathcal{W}$ that orders conceptual values in abstractness.
3. An interpretation function $I : \mathcal{U} \rightarrow \mathcal{W}$ that satisfies:
    a. For each attribute type τ and values w_1 and w_2 in $[\![\tau]\!]$ it holds that $w_1 = w_2$ if there is a conceptual value $z \in \mathcal{W}$ such that $z \leq_{\mathcal{W}} I(w_1)$ and $z \leq_{\mathcal{W}} I(w_2)$.
    b. If $\tau < \sigma$ then for every $w_1 \in [\![\tau]\!]$ and $w_2 \in [\![\sigma]\!]$ it holds that $K[\tau < \sigma](w_1) = (w_2)$ iff $I(w_1) \leq_{\mathcal{W}} I(w_2)$.

The requirement that there must be a valid interpretation can be restated as a set of constraints that must hold for the lattice, the associated strict semantics and the coercion functions. These are the following constraints:

- **ST1:** If there is an attribute value $w \in [\![\tau]\!] \cap [\![\sigma]\!]$ then $K[\tau < \sigma](w) = w$, i.e., every value that appears in both the subtype and in the supertype is mapped by the coercion

function to itself.

- **ST2:** If there is (1) an attribute value $w \in [\![\tau_1]\!] \cap [\![\sigma_1]\!]$, (2) two coercion functions $K[\tau_1 < \tau_2]$ and $K[\sigma_1 < \sigma_2]$, and (3) a type τ' such that $K[\tau_1 < \tau_2](w) \in [\![\tau']\!]$ and $K[\sigma_1 < \sigma_2](w) \in [\![\tau']\!]$, then $K[\tau_1 < \tau_2](w) = K[\sigma_1 < \sigma_2](w)$, i.e., if two coercion functions map the same value to two values that appear together in the strict semantics of some type, then the results of the two coercion functions are identical.
- **ST3:** If there is an attribute value $w \in [\![\tau]\!] \cap [\![\sigma]\!]$ and a type τ' such that $\tau < \tau' < \sigma$ then $w \in [\![\tau']\!]$, i.e., if a value appears in the subtype and the supertype then it also appears in all the types between the subtype and the supertype.

Indeed, it can be shown that these constraints characterise the existence of a valid interpretation, i.e., they are both sufficient and necessary conditions for the existence of a valid interpretation.

Since we do not explicitly mention the coercion functions in our framework, the assumptions **ST1** and **ST2** cannot be directly stated, but they do at this level imply the following constraint:

- **ST4:**If a subtype has a non-empty strict semantics then the supertype also has a non-empty strict semantics.

We will not add **ST3** and **ST4** at this point to our standard assumptions because it contradicts the assumptions in [GQL:EWD], as is discussed in Section Additional assumptions in WG3:MMX-010r2. However, we do see them as valid assumptions under the presented interpretation of subtyping.

The Liskov substitution principle revisited

The notion of consistent behaviour, as discussed in the preceding sections, is a necessary condition for adhering to the Liskov substitution principle, but it is not a sufficient condition. A potential complication is that in the setting of GQL certain inputs may be typed or untyped, which can make the interpretation of the substitution principle less straightforward. For example, it might be expected the the principle implies that any operation with signature $\tau_1 \times \tau_2 \rightarrow \tau_3$ allows input values of subtypes of τ_1 and τ_2 but coerces them to the expected type. That would guarantee that if the operator is given as input a less abstract version of a value, the result is as if it had been fed the value at the expected abstraction level.

However, this might not always be appropriate in a partially untyped setting. To illustrate this, consider the following expression, where `x` refers to some element in a graph:

```
x.a = x.b
```

It seems reasonable to assume here that `=` has a signature anyPropVal × anyPropVal → Boolean. However, mapping the operands `x.a` and `x.b` to that abstraction level means that all records are mapped to the empty record, and therefore would all be considered equal. This is

of course not the most intuitive interpretation of the equality operator, which would be to simply test the equality of the operands. It is therefore explicitly allowed that operations with a certain signature do **not** coerce their input values to the expected type. This does not exclude that some operations will consistently coerce their input values. For example, there might be an explicitly typed equality operation that coerces all its input values to some explicitly specified (record) type and at that level compares the input values.

A similar problem occurs with simple operations like the addition. For example, assume we have the following integer types with the indicated subtype relation: **int8** < **int16** < **int32** < **int64**. Now consider the following expression:

```
x.a + x.b
```

It might be that for each integer type there is a specific addition that behaves somewhat differently from the others in its overflow behaviour: the **int8** addition might result in an overflow on certain integers, where the **int16** addition does not. This expression can however be given a consistent interpretation that satisfies the substitution principle by assuming it coerces the input values to the smallest type of integer that avoids the overflow (if it can) and executes the corresponding addition. So that would mean here (assuming **int64** is the largest type to which the addition applies) that the addition behaves in a manner that is consistent as if it coerces all input values to **int64**. Also here there can be an option of a typed addition where the programmer explicitly indicates the input type.

Weak and strong conformance for types

The strict semantics of a type, denoted as $[\![\tau]\!]$, will later be used to define strict conformance for schemas. In addition we define a semantics that takes subtyping into account and that will be used for defining (weak) conformance. For each $\tau \in \mathcal{T}_{attr}$ we define a set $[\![\tau]\!]^* \subseteq \mathcal{V}$ that contains exactly all attribute values that conform to τ, by which we mean here all attribute values that strictly conform to a subtype of τ. More formally: $[\![\tau]\!]^* = \cup_{\sigma \leq \tau} [\![\sigma]\!]$. We call $[\![\tau]\!]^*$ the weak semantics of τ or also simply the semantics of attribute type τ. This approach to defining the semantics of subtyping gives an interpretation to subtyping that flexible, powerful and both informally and formally easy to understand][Balsters:1991].

We will call a type τ an abstract type if it holds that $[\![\tau]\!]^* = \cup_{\sigma < \tau} [\![\sigma]\!]^*$. If a type is not an abstract type we will call it a concrete type. Note that this deviates from the definition in [GQL:EWD] where an abstract type is defined as a type where the strict semantics is empty. We will come back to this in Section An analysis of the assumptions in [GQL:EWD].

Discussion of the assumptions for attribute types

The definitions and assumptions stated for the type lattice are sufficient to proceed with the presentation of the semantics of graph schemas, but several additional assumptions can be considered to make it easier to implement and understand. We briefly discuss some of them.

Union and intersection types

The lattice operations are similar to union and intersection types, but do not necessarily have all the properties that one would expect of them. For example, the following equalities do not follow from the preceding definitions and assumptions:

- **A1:** $[\![\tau \wedge \sigma]\!]^* = [\![\tau]\!]^* \cap [\![\sigma]\!]^*$
- **A2:** $[\![\tau \vee \sigma]\!]^* = [\![\tau]\!]^* \cup [\![\sigma]\!]^*$

It seems reasonable to add these as assumptions, which then justifies identifying the lattice operations with union types and intersection types. Note that **A2** is equivalent to assuming that $\tau \vee \sigma$ is an abstract type, since all strict subtypes of $\tau \vee \sigma$ must be subtypes of τ or subtypes of σ.

It can be shown that **A2** follows from the definitions in [GQL:EWD], because it introduces an explicit union type constructor, which we denote as $+$, and for which it holds by definition that $[\![\tau \mid \sigma]\!]^* = [\![\tau]\!]^* \cup [\![\sigma]\!]^*$. Since the document also assumes that lattice subtyping and extensional subtyping coincide, it follows from the fact that $[\![\tau \mid \sigma]\!]^* = [\![\tau]\!]^* \cup [\![\sigma]\!]^*$ is the smallest superset of both $[\![\tau]\!]^*$ and $[\![\sigma]\!]^*$, that $\tau \mid \sigma$ is the smallest supertype of both τ and σ, which by definition is equal to $\tau \vee \sigma$. It follows that $[\![\tau \vee \sigma]\!]^* = [\![\tau \mid \sigma]\!]^* = [\![\tau]\!]^* \cup [\![\sigma]\!]^*$.

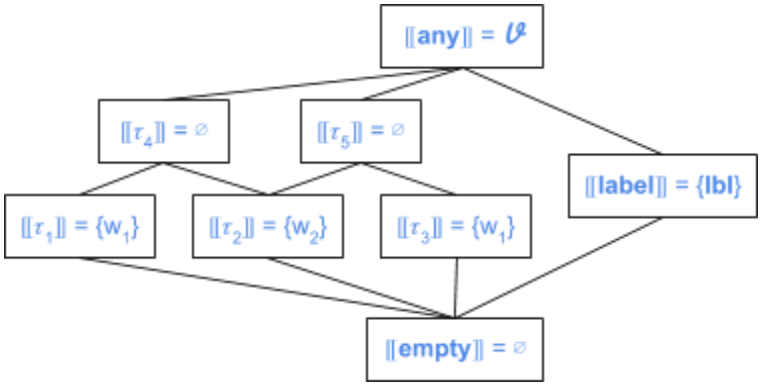
As will be discussed in the Section Lattice subtyping versus extensional subtyping, also **A1** can be shown to follow from the assumptions in [GQL:EWD].

Note that **A1** and **A2** do not imply that $[\![\tau \vee \sigma]\!] = [\![\tau]\!] \cup [\![\sigma]\!]$, which also would be an appropriate assumption to add for union types. However, this would contradict some assumptions in [GQL:EWD], as discussed in Section An analysis of the assumptions in [GQL:EWD].

Lattice subtyping versus extensional subtyping

The given definitions allow us to compare types in two different ways: with the lattice order $\tau \leq \sigma$, and with the order defined by the set semantics $[\![\tau]\!]^* \subseteq [\![\sigma]\!]^*$. Indeed, in [GQL:EWD] it is stated that this holds by definition: “Given two data types *T* and *U*, *T* is a subtype of *U* if and only if every element of *T* is also an element of *U*.” **(4.12.3.1 Subtype relation, p. 51)**q. It does follow that $[\![\tau]\!]^* \subseteq [\![\sigma]\!]^*$ if $\tau \leq \sigma$, but the reverse does not necessarily hold. Consider the following counterexample:

- $\mathcal{T}_{attr} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \textbf{label}, \textbf{empty}, \textbf{any}\}$,
  $\mathcal{V} = \{w_1, w_2, \textbf{lbl}\}$
- $[\![\tau_1]\!] = \{w_1\}$, $[\![\tau_2]\!] = \{w_2\}$, $[\![\tau_3]\!] = \{w_1\}$, $[\![\tau_4]\!] = \varnothing$, $[\![\tau_5]\!] = \varnothing$,
  $[\![\textbf{label}]\!] = \{\textbf{lbl}\}$, $[\![\textbf{empty}]\!] = \varnothing$, $[\![\textbf{any}]\!] = \mathcal{V}$,
- $\textbf{empty} < \tau_1$, $\textbf{empty} < \tau_2$, $\textbf{empty} < \tau_3$,
  $\tau_1 < \tau_4$, $\tau_2 < \tau_4$, $\tau_2 < \tau_5$, $\tau_3 < \tau_5$, $\tau_4 < \textbf{any}$, $\tau_5 < \textbf{any}$,
  $\textbf{empty} < \textbf{label} < \textbf{any}$



Here we have $[[\tau_4]]^* = \{w_1, w_2\}$ and $[[\tau_5]]^* = \{w_1, w_2\}$ but it does not hold that $\tau_4 \leq \tau_5$.

To prevent this situation, we can add the following assumption:

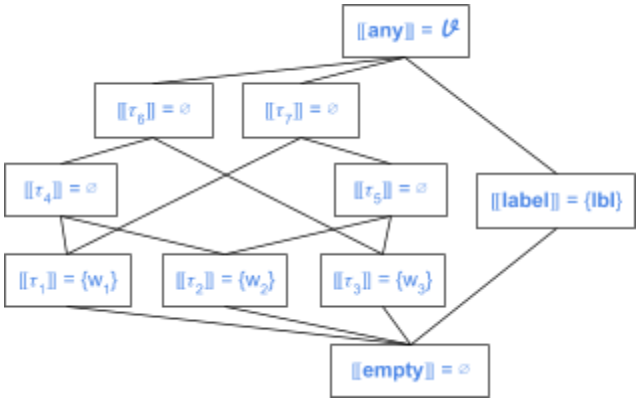
- **A3:** $[[\tau]] \cap [[\sigma]] \subseteq [[\tau \wedge \sigma]]^*$

Intuitively **A3** states that if two types share a value in their strict semantics this must be because they "inherit" this value from a common subtype. In other words, for each attribute value w there is a unique smallest type τ such that $w \in [[\tau]]$. We will refer to this type as **the most specific type of** w. Note that the previous counterexample indeed violated **A3** since w_1 does not have such a most specific type. Unfortunately **A3** is not sufficient, as is illustrated by the following counterexample:

- $\mathcal{T}_{attr} = \{\tau_1, \tau_2,$ **label**, **empty, any**$\}$,
  $\mathcal{V} = \{w_1,$ **lbl**$\}$
- $[[\tau_1]] = \{w_1\}$, $[[\tau_2]] = \varnothing$, $[[$**label**$]] = \{$**lbl**$\}$, $[[$**empty**$]] = \varnothing$, $[[$**any**$]] = \mathcal{V}$,
- **empty** $< \tau_1 < \tau_2<$ **any, empty** $<$ **label** $<$ **any**

This lattice satisfies **A3**, but we have $[[\tau_1]]^* = \{w_1\}$ and $[[\tau_2]]^* = \{w_1\}$ while it does not hold that $\tau_1 \leq \tau_2$. This counterexample is a bit strange in that it has an abstract type τ_2 which is not a consequence of a union type. Another counterexample shows that even if we only allow such abstract types, there is still a problem.

- $\mathcal{T}_{attr} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7,$ **label**, **empty, any**$\}$,
  $\mathcal{V} = \{w_1, w_2, w_3,$ **lbl**$\}$
- $[[\tau_1]] = \{w_1\}$, $[[\tau_2]] = \{w_2\}$, $[[\tau_3]] = \{w_3\}$, $[[\tau_4]] = \varnothing$, $[[\tau_5]] = \varnothing$, $[[\tau_6]] = \varnothing$, $[[\tau_7]] = \varnothing$,
  $[[$**label**$]] = \{$**lbl**$\}$, $[[$**empty**$]] = \varnothing$, $[[$**any**$]] = \mathcal{V}$,
- **empty** $< \tau_1$, **empty** $< \tau_2$, **empty** $< \tau_3$,
  $\tau_1 < \tau_4$, $\tau_2 < \tau_4$, $\tau_2 < \tau_5$, $\tau_3 < \tau_5$, $\tau_4 < \tau_6$, $\tau_3 < \tau_6$, $\tau_1 < \tau_7$, $\tau_5 < \tau_7$,
  $\tau_6 <$ **any**, $\tau_7 <$ **any**,
  empty $<$ **label** $<$ **any**



Here we have $[[\tau_6]]^* = \{w_1, w_2, w_3\}$ and $[[\tau_7]]^* = \{w_1, w_2, w_3\}$ but it does not hold that $\tau_6 \leq \tau_7$.

To prevent the problems of the previous two counterexamples, we add a final assumption which states that we can compare types by comparing their set of concrete subtypes.

- **A4:** Two types with the same set of concrete subtypes are identical.

This prevents the previous counterexample, because there we have two types, namely τ_1 and τ_2, that have the same concrete subtypes, namely the set $\{\tau_1\}$, but are not identical. Together, **A3** and **A4** are sufficient to prove that $[[\tau]]^* \subseteq [[\sigma]]^*$ iff $\tau \leq \sigma$.

Moreover, it can be shown that the addition of **A3** allows us to prove **A1**. This is because we can derive from the lattice assumptions that $[[\tau \wedge \sigma]]^* \subseteq [[\tau]]^* \cap [[\sigma]]^*$. Moreover, for any value $w \in [[\tau]]^* \cap [[\sigma]]^*$ it holds that $w \in [[\tau']]$ and $w \in [[\sigma']]$ for some type $\tau' \leq \tau$ and for some type $\sigma' \leq \sigma$. It then follows by **A3** that $w \in [[\tau' \wedge \sigma']]^*$. From the lattice it follows that $(\tau' \wedge \sigma') \leq (\tau' \wedge \sigma')$, and so we can conclude that $w \in [[\tau \wedge \sigma]]^*$.

It can be observed that **A3** follows from the definitions and assumptions in [GQL:EWD]. More specifically, **A3** follows from "*The most specific type* of an element of a data type is the concrete existing data type that is a subset of all other existing data types that contain that element. The most specific type of an element of a data type is always uniquely defined." **(4.12.1 Instance conformance, p. 51)**.

Whether **A4** follows from the assumptions in [GQL:EWD] could not be verified at the time of writing.

An analysis of the assumptions in [GQL:EWD]

The strict semantics of a type is defined in [GQL:EWD] as follows: "An instance *V strictly conforms* to a data type *T* if and only if *V* conforms to *T* and there is no data type *U* that is a (non-empty) strict subtype of *T* such that *V* is also an element of *U*." **(4.12.2 Instance conformance, p. 51)** From this it follows that $[[\tau]] \cap [[\sigma]] = \emptyset$ if τ is a strict subtype of σ. However, a consequence of this is that the strict semantics of all abstract types is empty. This is because any value in their semantics (and so also in their strict semantics) must be in the strict semantics

of at least one of their strict subtypes. So it follows that $[\![\tau \vee \sigma]\!] = \varnothing$ for any two types τ and σ, and also that $[\![\textbf{anyPropValue}]\!] = \varnothing$, which means all such abstract types are rendered unusable under strict schema conformance.

Another argument against this assumption is that it precludes some quite reasonable overlap under strict semantics. For example, it would be intuitive and useful to let the strict semantics of `VARCHAR(10)` overlap with `VARCHAR(20)`, and let the strict semantics of `INTEGER ARRAY[10]` overlap with `INTEGER ARRAY[20]`. This is consistent with and would follow from the interpretation of `VARCHAR(10)` as *"all strings of up to 10 characters"*, and of `INTEGER ARRAY[10]` as *"all integer arrays with length up to 10"*. Not allowing this overlap would mean that under strict conformance for schemas, a property with a value of type `VARCHAR(10)` would not be allowed where a value of type `VARCHAR(20)` is expected. Of course this might also be dealt with through coercion, but if coercion does not change the value, just the associated type, then it effectively behaves as if there is overlap.

Another assumption in [GQL:EWD] is the following: "All data types that are the most specific type of some instance are disjoint and therefore two instances that do not have the same most specific type are never considered to be identical" **(4.12.1 General information about data types, p. 51)** The first part of this sentence can be stated as $[\![\tau]\!]^* \cap [\![\sigma]\!]^* = \varnothing$ if τ and σ are distinct concrete types. This might be regarded as too strict. For example, it disallows that the semantics of `{ name STRING }` overlaps with that of `{ name STRING, address STRING }`. As a consequence it would not be possible to let `{ name STRING, address STRING }` be a subtype of `{ name STRING }`. So under (weak) conformance for schemas, a property with a value of type `{ name STRING, address STRING }` would not be allowed where a value of type `{ name STRING }` is expected. Another problematic example would be user-defined subtypes of types like `VARCHAR(100)` defined by restrictions such as regular expressions. Under this assumption it would be impossible to have two such types with a third one being a common subtype. This is because the assumption says that the intersection of the semantics of the two super-types must be empty, and so the semantics of the subtype must also be empty.

The previously mentioned quote from [GQL:EWD] was perhaps meant to say something weaker than our interpretation. An alternative interpretation might be that $[\![\tau]\!] \cap [\![\sigma]\!] = \varnothing$ if τ and σ are distinct concrete types, so require that the strict semantics do not overlap rather than the weak semantics, which is a weaker assumption. However, this brings us back to the same issues that, at the beginning of this section, are associated with requiring that the strict semantics of a strict subtype cannot overlap with that of the supertype.

In [GQL:EWD] a statement is made concerning the cardinality of the extension of types, so about the sizes of $[\![\tau]\!]^*$. It states: "Given two data types *T* and *U*, if *T* is a strict subtype of *U* then *T* is smaller than *U*.". Here "smaller" refers to the cardinality being smaller. This can be correct, but does require the additional assumptions that (1) the set $\mathcal{A}$ of attribute names is finite and (2) the set $\mathcal{V}$ of attribute values is finite. If either of these sets is countably infinite, then the extension of a record type will be countably infinite (since it contains the extensions of countably

infinite subtypes) and therefore will all have the same cardinality. Since some of these record types are strict subtypes of other records types, this would contradict the claim that strict subtypes are strictly smaller. If we want to ensure that $\mathscr{U}$ is finite in a type system such as is proposed in [GQL:EWD] where there are atomic types, record types and collection types which can all be recursively nested, then it is not enough to require that all atomic types have finite extensions. In addition it should be required that there is an upper bound on the cardinality of values of a collection type, the width of records and the nesting depth of recursively nested attribute values.

On axiomatizing and deciding the subtype relationship

In [GQL:EWD] an extension-based definition of subtyping is given, and next to that also rules for deciding this relationship are given. This raises the issue of soundness and completeness for these rules, which can be sometimes hard to establish in the presence of union types, as discussed in [Hidders:1995]. For example, the following two rules are not sound:

- $\sigma \vee \tau \leq \tau'$ iff $\sigma \leq \tau'$ and $\tau \leq \tau'$
- $\tau' \leq \sigma \vee \tau$ iff $\tau' \leq \sigma$ or $\tau' \leq \tau$

As an example consider (**int** ∨ **bool**) ∨ **string** ≤ **int** ∨ (**bool** ∨ **string**), which clearly holds, but according to both rules is false. These rules are however sound if we assume that the types σ, τ and τ' are concrete types. In fact, the following more general rule is sound if we assume all σ_i, τ_j are concrete:

- $\sigma_1 \vee \ldots \vee \sigma_n \leq \tau_1 \vee \ldots \vee \tau_m$ iff for every σ_i there is a τ_j such that $\sigma_i \leq \tau_j$

This rule can provide a sound and complete axiomatisation if (1) union types are the only abstract types and (2) there is a sound and complete axiomatisation for concrete types. Usually, however, assumption (1) is not valid, for example if there are record types such as { name ↦ **string**, birthdate ↦ (**date** ∨ **null**) }, which is an abstract type. However, this is not a problem if we can assume that all such types can be rewritten to a finite union of concrete types. For example, the previous record type is equivalent to the union type { name ↦ **string**, birthdate ↦ **date** } ∨ { name ↦ **string**, birthdate ↦ **null** }.

It can be that there are abstract types that cannot be reduced to a finite union of concrete types. For example, there might be a type **string** with an infinite set of concrete subtypes **varchar**(1), **varchar**(2), et cetera. In that case it is sufficient to assume that the subtype relationship over concrete types and irreducible abstract types is axiomatizable. This is because the previous rule is in fact sound for the more general case where all σ_i, τ_j are irreducible, i.e, cannot be reduced to finite unions of concrete types, and so are either concrete types or irreducible abstract types.

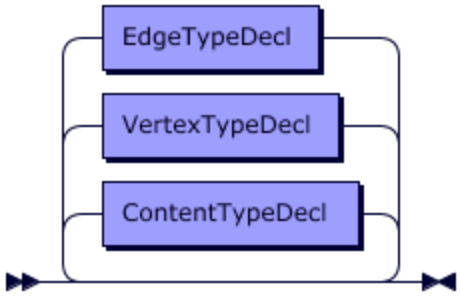
The syntax of graph-type cores

We start with a very simple type of graph-type core that only considers vertex types and vertex types and does not consider any form of inheritance or subtyping. We focus in this document on the graph-type core, i.e., the part of the schema that specifies the allowed types, and so leave aside schema constraints and graph-level attributes.

Concrete syntax

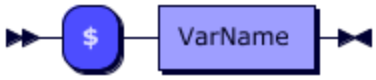
A graph-type core consists of a list of type specifications, which can be either vertex type specifications or vertex type specifications. A graph-type core specification has the following syntax:

SchemaCore ::= (ContentTypeDecl | VertexTypeDecl | EdgeTypeDecl)*

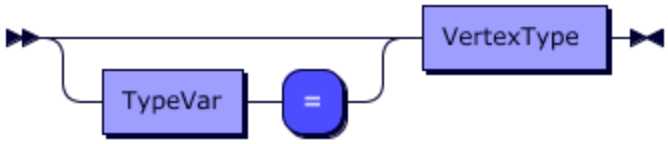


ContentTypeDecl ::= TypeVar "=" ContentType

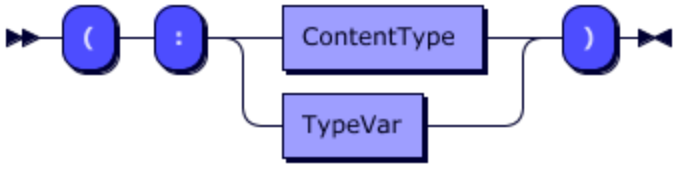
TypeVar ::= "$" VarName



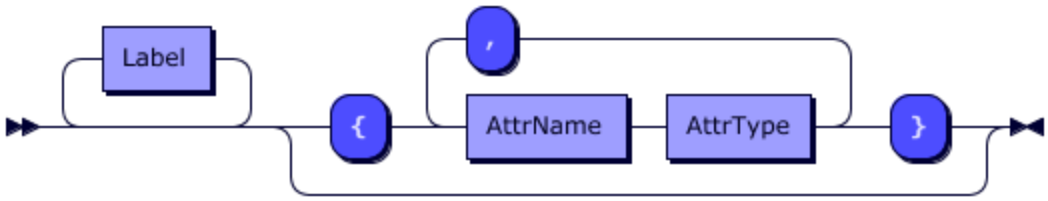
VertexTypeDecl ::= (TypeVar "=")? VertexType



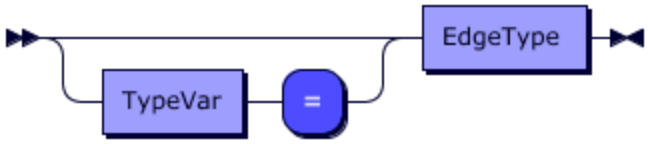
VertexType ::= "(" ":" (ContentType | TypeVar) ")"



ContentType ::= Label* ("{" AttrName AttrType ("," AttrName AttrType)* "}")?



EdgeTypeDecl ::= (TypeVar "=")? EdgeType



EdgeType ::= VertexType "-[" ":" ContentType "]->" VertexType

An example of a graph-type core is:

- `$personContent = :Person { name STRING, birthdate DATE }`
- `$person = (: $personContent )`
- `$city = (:City Place { name STRING, url URL })`
- `$country = (:Country Place { name STRING, url URL })`
- `$continent = (:Continent Place { name STRING, url URL })`
- `$livesIn = (:$person)-[:livesIn { start DATE }]->(:$city)`
- `$worksIn = (:$person)-[:worksIn { start DATE }]->(:$city)`
- `$cityLiesIn = (:$city)-[:liesIn]->(:$country)`
- `$countryLiesOn = (:$country)-[:liesOn]->(:$continent)`

The type names should be understood as a macro mechanism. So the above graph-type core is equivalent to:

- `(:Person { name STRING, birthdate DATE })`
- `(:City Place { name STRING, url URL })`
- `(:Country Place { name STRING, url URL })`
- `(:Continent Place { name STRING, url URL })`
- `(:Person { name STRING, birthdate DATE })`
  `-[:livesIn { start DATE}]->`
  `(:City Place { name STRING, url URL })`
- `(:Person { name STRING, birthdate DATE })`
  `-[:worksIn { start DATE}]->`
  `(:City Place { name STRING, url URL })`
- `(:City Place { name STRING, url URL })`
  `-[:liesIn]->`
  `(:Country Place { name STRING, url URL })`
- `(:Country Place { name STRING, url URL })`
  `-[:liesOn]->`

```
(:Continent Place { name STRING, url URL })
```

Since labels are regarded as a special type of attribute, their denotation can be understood as a short-hand for denoting attributes of type **label**, which in the concrete syntax we will denote as `LABEL`. So, for example the content type

```
City Place { name STRING, url URL }
```

could also have been written as

```
{ City LABEL, Place LABEL, name STRING, url URL }
```

Note: Under the approach that was suggested earlier where we distinguish marked and unmarked attribute names to indicate properties and labels respectively, the alternative representation for the content type would be:

```
{:City LABEL, :Place LABEL, name STRING, url URL}
```

End note

We can allow using `&` between labels. So the content type `City Place { name STRING, url URL }` can also be written as `City & Place { name STRING, url URL }`. We expect this to be consistent with a later generalization where `&` is defined to be the type intersection operator. So the content type might then for example also be written as `City & Place & { name STRING } & { url URL }`.

Abstract syntax

We start with defining the auxiliary concept of content type, which describes the attributes of an element.

Definition: A content type is a finite partial function $\kappa : \mathcal{A} \rightharpoonup \mathcal{T}_{attr}$ that maps some attribute names to an attribute type.

Example
An example of a content type is function κ such that

- κ(Person) = **label**
- κ(birthdate) = **date**
- κ(name) = **string**

- κ(startDate) = **date**

We will denote this κ = { Person ↦ **label**, birthdate ↦ **date**, name ↦ **string**, startDate ↦ **date** }.

End example

A graph-type core is formally defined as follows.

Definition: A graph-type core is defined as a set of vertex types where a vertex type is defined as either a simple vertex type, denoted as **vertex**(κ) where κ is a content type, or a vertex type, denoted as **edge**(σ, κ, τ) where σ is a simple vertex type that describes the tail vertex, κ is a content type that describes the attributes of the edge and τ is a content type that describes the head vertex. In each case we refer to κ as the content type of the vertex type. The set of all simple vertex types is denoted as $\mathcal{T}_{vertex}$, and the set of all vertex types is denoted as $\mathcal{T}_{edge}$.

Graph-type core Example

Consider the following graph-type core:

- `(:Person { name STRING, birthdate DATE })`
- `(:City { name STRING, url URL })`
- ```
  (:Person { name STRING, birthdate DATE })
    -[:worksIn { start DATE}]->
  (:City { name STRING, url URL })
  ```

The following is the graph-type core represented in the formal notation from the definition:
C = { **vertex**(κ_1), **vertex**(κ_2), **edge**(**vertex**(κ_1), κ_3, **vertex**(κ_2)) } where

- κ_1 = { Person ↦ **label**, fullName ↦ **string**, birthdate ↦ **date** }
- κ_2 = { City ↦ **label**, name ↦ **string**, url ↦ **url** }
- κ_3 = { worksIn ↦ **label**, start ↦ **date** }

End example

Mapping the concrete syntax to the abstract syntax

The mapping of concrete syntax to the abstract syntax is as follows. We start with rewriting the concrete syntax to a normal form where no variables are left. We do this in two steps:

1. We iterate over the type declarations that define a variable, in the order that they appear in the graph-type core expression. For each content and vertex type declaration we apply it as a substitution to all the subsequent type declarations up to the end of the
  ```

graph-type core expression or a type declaration that redefines the same variable, whichever comes first. The substitution replaces the variable with the content type assigned to the variable, or the content type of the vertex type assigned to the variable, where the latter might itself be a variable. If the variable is bound to an edge type then we do not apply it as a substitution.

2. We remove any type declarations that still have variables left in them. We also remove any content declarations. The result of this step is a list of element type declarations without variables in them.
3. We replace all explicit labels with equivalent attribute specifications in the specified content type.

The resulting list of element types is then mapped to a set of formal element types by mapping each specified type to its formal counterpart in the obvious manner: each vertex type is mapped to a formal edge type with the specified content type and each edge type is mapped to a formal edge type with tail content type, content type and head content type as specified in the expression.

The semantics of graph-type cores

In this section we will present the proposals for the semantics of graph-type cores. We will start with the semantics that is considered preferable by most members of GS-Basic. This is followed by the alternatives that were also considered and even suggested by some members.

The at-least-one-match semantics

The semantics of a graph-type core is defined by defining when a certain property graph conforms to a certain graph-type core. This is in turn based on the notion of matching, that defines when a certain element in a graph matches a certain type, which roughly means that all attributes required by the type are present in the element and contain a value of the correct type. We will distinguish two kinds of matching: *exact matching*, which requires that the type describes all attributes of the element, and *over matching*, which does not require this (and so allows the element to have more attributes than are specified by the type) . We start with defining these notions for content types:

Definition: Given a content type κ we define the set of all over matches of κ, denoted as $[\![\kappa]\!]^*$, is defined as all finite partial functions $r : (\mathcal{A} \rightharpoonup \mathcal{V})$ such that for every $(a \mapsto \sigma) \in \kappa$ there is a pair $(a \mapsto w) \in r$ such that $w \in [\![\sigma]\!]^*$. The set of all exact matches of κ, denoted as $[\![\kappa]\!]$, is defined as all $r \in [\![\kappa]\!]^*$ such that for every pair $(a \mapsto w) \in r$ there is a pair $(a \mapsto \sigma) \in \kappa$ such that $w \in [\![\sigma]\!]$.

For example, consider the content type { street ↦ **string**, city ↦ **string** } then

- the record { street ↦ *"Malet st"*, city ↦ *"London"* } is an over match and also an exact match because the attributes *street* and *city* are defined and only those attributes and nothing else.
- the record { street ↦ *"Malet st"*, city ↦ *"London"*, country ↦ *"UK"* } is an over match because the attributes *street* and *city* are defined but not an exact match because it has the attribute *country* which is not defined in the content type.
- the record { street ↦ *"Malet st"* } is neither an over match nor an exact match because it is missing the attribute *city*, and
- the record { street ↦ *5*, city ↦ *"London"*, country ↦ *"UK"* } is neither an over match nor an exact match because the value for the attribute *street* is a value that does not match the string type.

The notions of subtyping and intersection types, which is postulated for attribute types, can be generalized for content types as follows. If we have two content types κ_1 and κ_2 we say that κ_1 is a subtype of κ_2, denoted as $\kappa_1 \leq \kappa_2$, if for every pair $(a \mapsto \tau) \in \kappa_2$ there is a pair $(a \mapsto \sigma) \in \kappa_1$ such that $\sigma \leq \tau$. The implied intersection type can be formulated as follows. If we have two content types κ_1 and κ_2 then $\kappa_1 \wedge \kappa_2$ denotes the content type κ_3 where $\mathbf{dom}(\kappa_3) = \mathbf{dom}(\kappa_1) \cup \mathbf{dom}(\kappa_2)$ and

- for $a \in \mathbf{dom}(\kappa_1) \setminus \mathbf{dom}(\kappa_2)$ holds $\kappa_3(a) = \kappa_1(a)$
- for $a \in \mathbf{dom}(\kappa_1) \cap \mathbf{dom}(\kappa_2)$ holds $\kappa_3(a) = \kappa_1(a) \wedge \kappa_2(a)$
- for $a \in \mathbf{dom}(\kappa_2) \setminus \mathbf{dom}(\kappa_1)$ holds $\kappa_3(a) = \kappa_2(a)$.

Note that these notions of subtyping and intersection types are consistent in the sense that the intersection type identifies the greatest common subtype of two types, and that they satisfy the relationship $[\![\kappa_1 \wedge \kappa_2]\!]^* = [\![\kappa_1]\!]^* \cap [\![\kappa_2]\!]^*$. Moreover, it is consistent with the "is of type" definition for attribute values, which means here that it holds that $[\![\kappa]\!]^* = \cup_{\kappa' \leq \kappa} [\![\kappa']\!]$.

Based on the preceding notions we generalise the notions of matching for the level of elements:

Definition: Given a property graph $G = (V, E, \varrho, \alpha)$ we define the set of over matches in G of a simple type τ, denoted as $[\![\tau]\!]^*_G$, is defined as all elements $x \in V \cup E$ such that

- if κ is the content type of τ then $\alpha(x) \in [\![\kappa]\!]^*$, and
- if $\tau = \mathbf{edge}(\mathbf{vertex}(\kappa_1), \kappa, \mathbf{vertex}(\kappa_2))$ then for all vertices v_1 and v_2 such that $(v_1, v_2) = \varrho(x)$ it holds that $\alpha(v_1) \in [\![\kappa_1]\!]^*$ and $\alpha(v_2) \in [\![\kappa_2]\!]^*$.

The set of exact matches in G of a simple type τ, denoted as $[\![\tau]\!]_G$, is defined as all elements $x \in V \cup E$ such that:

- if κ is the content type of τ then $\alpha(x) \in [\![\kappa]\!]$, and
- if $\tau = \mathbf{edge}(\mathbf{vertex}(\kappa_1), \kappa, \mathbf{vertex}(\kappa_2))$ then for all vertices v_1 and v_2 such that $(v_1, v_2) = \varrho(x)$ it holds that $\alpha(v_1) \in [\![\kappa_1]\!]$ and $\alpha(v_2) \in [\![\tau_2]\!]$.

Example

Consider the following graph-type core:

- `(:Person { name STRING, birthdate DATE })`
- `(:City { name STRING, url URL })`
- `(:Person { name STRING, birthdate DATE })`
  `-[:worksIn { start DATE}]->`
  `(:City { name STRING, url URL })`

Let's go through the following vertices and edges:

- `(: {Person=lbl, name="Jan Hidders"})`
  - **over match:** No
  - **exact match:** No
- `(: {Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  - **over match**: Yes
  - **exact match**: Yes
- `(: {Person=lbl, name="Jan Hidders", birthdate="1980-01-01", birthplace="Deventer"})`
  - **over match**: Yes
  - **exact match**: No
- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  `(v2 :{City=lbl, name="London", url="www.london.org"})`
  `(v1) -[e1 :worksIn]-> (v2)`
  - **over match (of edge** e1**)**: No
  - **exact match (of edge** e1**)**: No
- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  `(v2 :({City=lbl, name="London", url="www.london.org"})`
  `(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)`
  - **over match (of edge** e1**)**: Yes
  - **exact match (of edge** e1**)**: Yes
- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  `(v2 :{City=lbl, name="London", url="www.london.org"})`
  `(v1) -[e1 :worksIn {start="2020-01-01", foo= "bar"}]-> (v2)`
  - **over match (of edge** e1**)**: Yes
  - **exact match (of edge** e1**)**: No

End example

The notions of subtyping and intersection types, which we defined for content types, can be generalized for element types as follows. If we have two simple vertex types **vertex**(κ_1) and **vertex**(κ_2) then we say that **vertex**(κ_1) is a subtype of **vertex**(κ_2), denoted as **vertex**(κ_1) ≤ **vertex**(κ_2), if $\kappa_1 \leq \kappa_2$. Moreover, we say that **edge**(σ, κ, τ) is a subtype of **edge**(σ', κ', τ'), denoted

as **edge**$(\sigma, \kappa, \tau) \leq$ **edge**$(\sigma', \kappa', \tau')$, if $\sigma \leq \sigma'$, $\kappa \leq \kappa'$, and $\tau \leq \tau'$. The implied intersection type can be formulated as follows:

- **vertex**$(\kappa_1) \wedge$ **vertex**$(\kappa_2) =$ **vertex**$(\kappa_1 \wedge \kappa_2)$
- **edge**$(\sigma, \kappa, \tau) \wedge$ **edge**$(\sigma', \kappa', \tau') =$ **edge**$(\sigma \wedge \sigma', \kappa \wedge \kappa', \tau \wedge \tau')$

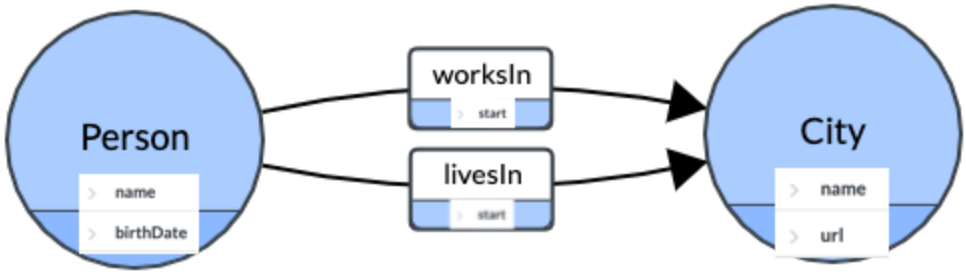
Note that also in this case these notions of subtyping and intersection types are consistent in the sense that the intersection type identifies the greatest common subtype of two types. Moreover, it is consistent with the “is of type” definition for attribute values, which means here that it holds for every vertex type τ that $[\![\tau]\!]^* = \cup_{\sigma \leq \tau} [\![\sigma]\!]$.

Finally we define when a property graph conforms to a graph-type core.

Definition: We say that a property graph $G = (V, E, \varrho, \alpha)$ conforms to a graph-type core C if for every element $x \in V \cup E$ there is a vertex type $\tau \in C$ such that $x \in [\![\tau]\!]^*_G$. We say that G strictly conforms to C if for every element $x \in V \cup E$ there is a vertex type $\tau \in C$ such that $x \in [\![\tau]\!]_G$.

Example

Consider the following graph-type core:



Given the following Property Graph G1:

```
(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})
(v2 :{City=lbl, name="London", url="www.london.org"})
(v3 :{City=lbl, name="Brussels", url="www.brussels.org"})
(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)
(v1) -[e2 :livesIn {start="2015-01-01"}]-> (v3)
```

Property Graph G1 strictly conforms to the graph-type core because every element is an exact match

Given the following Property Graph G2:

```
(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01",
birthplace="Deventer"})
(v2 :{City=lbl, name="London", url="www.london.org"})
(v3 :{City=lbl, name="Brussels", url="www.brussels.org"})
(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)
(v1) -[e2 :livesIn {start="2015-01-01"}]-> (v3)
```

Property Graph G2 conforms to the graph-type core because the vertex v1 is an over match.

Given the following Property Graph G3:

```
(v1 :{Person=lbl, name="Jan Hidders"})
(v2 :{City=lbl, name="London", url="www.london.org"})
(v3 :{City=lbl, name="Brussels", url="www.brussels.org"})
(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)
(v1) -[e2 :livesIn {start="2015-01-01"}]-> (v3)
```

Property Graph G3 does NOT conform (hence does not strictly conform) to the graph-type core because the vertex v1 is not an over match.

End example

The combinatorial semantics

Next to the basic notion of conformance that was just defined we also introduce a more liberal notion of conformance that we will call *combinatorial conformance* and that was in earlier discussions referred to as the type-1 semantics (as for example presented in "GS Basic, status update 23 March 2020" [Basic:23-03-2020]). Informally it can be described as allowing elements if all their attributes are each justified by at least one of the types they conform to, or equivalently, that their contents exactly match the intersection of the content types of the type they conform to.

Definition: We say that a property graph $G = (V, E, \varrho, \alpha)$ combinatorially conforms to a graph-type core C if for every element $x \in V \cup E$ it holds that (1) there is at least one element type $\tau \in C$ such that $x \in [\![\tau]\!]^*_G$ and (2) for every pair $(a \mapsto w) \in \alpha(x)$ there is at least one element type $\tau \in C$ with content type κ such that (a) $x \in [\![\tau]\!]^*_G$ and (b) there is a pair $(a \mapsto \sigma) \in \kappa$ such that $w \in [\![\sigma]\!]$.

Note that he previously definition can be equivalently stated as requiring that for every element $x \in V \cup E$ it holds that $\alpha(x) \in [\![\kappa_1 \wedge \ldots \wedge \kappa_n]\!]$ where $\{\kappa_1, \ldots, \kappa_n\}$ is the set of all content types of

element types τ in C such that $x \in [\![\tau]\!]^*_G$.

Example

Consider the following graph-type core:

```
(:Person { name STRING, birthdate DATE })
(:Prof { status STRING, university STRING })
```

And consider the following Property Graph with only one vertex:

```
(v1 :{Person=lbl, Prof=lbl, name="Jan Hidders", birthdate="1980-01-01",
status="Lecturer", university="Birkbeck"})
```

- **Combinatorial Conform:** Yes because the vertex has all the content types for `Person` and `Prof` hence it is an over match. Additionally, it only has the content types of both `Person` and `Prof`, therefore it is also an exact match
- **Person conform:** Yes because it has the attributes for `Person` (`name` and `birthdate`) and others (status and university)
- **Prof conform:** Yes because it has the attributes for `Prof` (`status` and `university`) and others (`name` and `birthdate`)
- **Person strictly conform:** No because it has more attributes than defined for `Person`
- **Prof strictly conform:** No because it has more attributes than defined for `Prof`

Now consider the following Property Graph with only one vertex:

```
(v1 :{Person=lbl, Prof=lbl, name="Jan Hidders", birthdate="1980-01-01",
status="Lecturer", university="Birkbeck", foo="bar"})
```

- **Combinatorial Conform:** No because it has an attribute that is not defined in any of the types of which it is an over match: foo.
- **Person conform:** Yes because it has the attributes for `Person` (`name` and `birthdate`) and others (status and university and foo)
- **Prof conform:** Yes because it has the attributes for `Prof` (`status` and `university`) and others (`name` and `birthdate` and `foo`)
- **Person strictly conform:** No because it has more attributes than defined for `Person`
- **Prof strictly conform:** No because it has more attributes than defined for `Prof`

End example

The combinatorial semantics essentially say that the content type of an element should exactly match the intersection of the content types of all element types which the element over-matches. So, for edge types the head and tail types are ignored for the exact match. To understand why, consider the following example.

```
$message = (:Message { creationDate DATE, content STRING })
$comment = (:Comment { public BOOL })

(:$comment)-[:isReplyTo]->(:$message)
```

The intention of the combinatorial conformance is to interpret the semantics of types based on over matching rather than exact matching and to allow vertices to fully match with combinations of vertex types. In this case the type `$comment` can for example be combined with the type `$message`. Under this interpretation it makes sense to let the schema require for `isReplyTo` edges that the head also over-matches `$comment`. Therefore we allow that such edges end in vertices that fully match a combination of `$message` and `$comment`. Observe that if in the formal semantics we would have required for edges that they fully match the combination of a subset of the edge types, this would not have been allowed.

Comparing normal conformance and combinatorial conformance

The combinatorial conformance semantics can be understood in terms of the normal conformance semantics. We can for example expand a graph-type core by adding combinations of existing types, by applying exhaustively the following rule:

- **E1:** If the graph-type core contains the types τ and σ then add $\tau \wedge \sigma$.

We will call the result of this the *expanded graph-type core*.

However, it is unfortunately not true that a property graph conforms to an expanded graph-type core iff it combinatorially conforms to the original graph-type core. Consider again the message example:

```
$message = (:Message { creationDate DATE, content STRING })
$comment = (:Comment { public BOOL })

(:$comment)-[:isReplyTo]->(:$message)
```

As mentioned earlier, under combinatorial conformance a property graph can contain vertices that belong to the combination of `$message` and `$comment`. Moreover, such a vertex might be at the head or the tail of an `isReplyTo` edge.

Let us now consider the expansion of this graph-type core, which results in the addition of the following type:

```
$messageComment = (:Message Comment { creationDate DATE, content STRING,
                                      public BOOL })
```

Note that since there is only one edge type, no new combinations of edge types are added. The resulting expanded schema, under normal conformance semantics, does allow vertices that belong to the intersection of `$message` and `$comment`, for which it has the `$messageComment` type. However, these vertices are under normal conformance semantics not allowed to participate in `isReplyTo` edges. Therefore the expanded graph-type core is not equivalent to the original graph-type core under combinatorial conformance semantics, since such vertices would there be allowed to participate in these edges.

However, this can be resolved by adding an additional expansion rule:

- **E2:** If the graph-type core contains **edge**(σ, κ, τ) and a vertex type τ', then add the types **edge**($\sigma \wedge \tau$', κ, τ) and **edge**(σ, κ, $\tau \wedge \tau$').

This rule will in the example add the following additional edge types:

```
(:$messageComment) -[:isReplyTo]-> (:$message)
(:$comment) -[:isReplyTo]-> (:$messageComment)
(:$messageComment) -[:isReplyTo]-> (:$messageComment)
```

After this we have indeed obtained a graph-type core that is under normal conformance semantics equivalent to the original graph-type core under combinatorial conformance semantics. Indeed, with this additional rule added to the exhaustive expansion process, it holds that every property graph combinatorially conforms to a graph-type core if and only if it conforms to the expansion of that graph-type core.

The exactly-one-match semantics

One might consider making the definition of conformance more strict and require that for each element there is exactly one fully matching type in the graph-type core. The formal definition for conformance would be changed to the following (with changes highlighted in grey).

> **Definition:** We say that a property graph $G = (V, E, \varrho, \alpha)$ conforms to a graph-type core C if for every element $x \in V \cup E$ there is exactly one vertex type $\tau \in C$ such that $x \in [\![\tau]\!]^*_G$. We say that G strictly conforms to C if for every element $x \in V \cup E$ there is exactly one vertex type $\tau \in C$ such that $x \in [\![\tau]\!]_G$.

However, that changes the semantics. To illustrate this, consider the following graph-type core;

- `$personWithLongName = (:Person { name VARCHAR(50) })`
- `$personWithShortName = (:Person { name VARCHAR(15) })`

Under the at-least-one-match semantics this has the following strictly conforming graph:.

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Person { name="George Walsh" }`

However, under exactly-one-match semantics this graph no longer strictly conforms, since `v2` matches both `$personWithLongName` and `$personWithShortName`. Similar behaviour occurs with other property value types that can overlap, like record types with optional attributes, or union types. These cannot be removed by rewriting if they are nested inside collection types.

On the other hand, if it holds that all attribute types are disjoint, i.e, for all two distinct attribute types τ_1 and τ_2 it holds that $[\![\tau_1]\!] \cap [\![\tau_2]\!] = \varnothing$, then the exactly-one-match semantics and the at-least-one-match semantics are equivalent for strict conformance.

This issue also manifests itself for weak conformance. Consider the following graph-type core:

```
$personWithAddress = (:Person Local { address { street STRING,
                                                number STRING,
                                                city STRING })
$personWithIntlAddress = (:Person { address { street STRING,
                                              number STRING,
                                              city STRING,
                                              country STRING })
```

Under weak conformance as defined by the exactly-one-semantics the following graph would not weakly conform.

```
(v1 :Person Local { street="Atomiumplein", number="1", city="Brussels",
country="Belgium" })
```

This is because it weakly conforms to both types. Also here we can observe that the at-least-one-match semantics and the exactly-one-match semantic are equivalent if we assume that there is no attribute value that weakly conforms to two distinct attribute types. However, this assumption does not hold if we allow subtyping for record types.

The isolation-aware semantics

The at-least-one-match semantics has an implicit assumption that the content types in an edge type in a graph-type core that describe the head and tail vertices correspond to the content type of a vertex type in the graph-type core. We will call the content types for which this is **not** the case dependent content types, and the vertex types they define dependent vertex types. If a graph-type core has dependent content types then under the current semantics the edge types in which they appear cannot be populated. For example, consider the following graph-type core declaration:

- `$PersonContent = Person { name STRING, birthdate DATE }`
- `$worksInContent = worksIn { start DATE}`
- `$CityContent = City { name STRING, url URL }`
- `(: $PersonContent )-[: $worksInContent ]-> (: CityContent )`

Note that in this declaration there are no vertex type declarations, and after substitution we obtain the following graph-type core:

- ```
  (:Person { name STRING, birthdate DATE })
    -[:worksIn { start DATE}]->
  (:City { name STRING, url URL })
  ```

In this case only the empty graph conforms, since the graph-type core contains no vertex types and so no graph containing vertices can conform.

One way to solve this is to allow vertices to exist already if they participate in an edge that is justified to exist, i.e., matches one of the edge types. We illustrate here how to adapt the definition for strict conformance, but the same principles can be applied to conformance and combinatorial conformance.

We would adapt the definition of strict conformance as follows: (changed parts in grey):

Definition: We say that G strictly conforms to C if

- for every edge $e \in E$ there is a vertex type $\tau \in C$ such that $e \in [\![\tau]\!]_G$, and
- for every vertex $v \in V$ that is not incident to any edge in G there is a simple vertex type $\tau \in C$ such that $v \in [\![\tau]\!]_G$.

Under these semantics the following graph strictly conforms to the preceding graph-type core.

- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
- `(v2 :{City=lbl, name="London", url="www.london.org"})`
- `(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)`

The next graph however does not strictly conform because it contains an isolated vertex without having a matching independent type in the graph-type core.

- `(v1 :{Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`

This change in semantics has advantages and disadvantages:

- **Good:** It adds expressive power and gives a meaningful semantics for graph-type cores that otherwise would only allow the empty graph.
  ```

- **Good:** It is similar to existing notions in conceptual data models such as ORM, where object types have to be declared explicitly as *independent* if their instances should be able to exist without participating in any non-identifying relationships.
- **Bad**: It makes the semantics a bit more complex and implies sophisticated database constraints that might not be obvious.
- **Bad**: In the case of conformance it might make certain forms of factorisation harder.

To illustrate the point about sophisticated database constraints, consider the following graph-type core:

- `(:Person { name STRING, birthdate DATE })`
  `-[:worksIn { start DATE}]->`
  `(:City { name STRING, url URL })`
- `(:Person { name STRING, birthdate DATE })`
  `-[:livesIn { start DATE}]->`
  `(:City { name STRING, url URL })`

Under the adapted semantics this implies the following two constraints (formulated in the syntax that is close the one that is discussed in the working group for keys and cardinality constraints):

```
WHERE (p :Person) REQUIRE
  (THERE IS AT LEAST ONE wi SUCH THAT (p)-[wi :worksIn]->()) OR
  (THERE IS AT LEAST ONE li SUCH THAT (p)-[li :livesIn]-())

WHERE (c :City) REQUIRE
  (THERE IS AT LEAST ONE wi SUCH THAT ()-[wi :worksIn]->(c)) OR
  (THERE IS AT LEAST ONE li SUCH THAT ()-[li :livesIn]-(c))
```

For the point about factorisation consider the following graph-type core:

- `(:person manager)`
- `(:person employee)`
- `(:person)-[:knows]->(:person)`

This would now allow vertices with just the label `person`, and not those with `manager` or `employee`, which might not be the intended meaning. We could try to remedy this by indicating that we allow subtypes, by for example replacing `:` with `<:`. So the edge type could then be:

- `(<:person)-[:knows]->(<:person)`

However, that would then allow too much, since we could have a vertex with any set of labels in such an edge, as long as the set of labels contains the label `person`.

The homomorphism-based semantics

An alternative semantics can be based on homomorphisms between property graphs and graph-type cores, as is proposed for example in [Bonifati:2019]. For the sake of this presentation we will discuss this in the context of strict conformance semantics, but the same principles can also be applied to conformance and combinatory conformance semantics.

Given a property graph a property graph $G = (V, E, \varrho, \alpha)$ and a graph-type core C we define a homomorphism as a function $h : (V \cup E) \rightarrow C$ such that for every element $x \in V \cup E$ it holds that x exactly matches $h(x)$.

Definition: Given a property graph a property graph $G = (V, E, \varrho, \alpha)$ and a graph-type core C we define a homomorphism from G to C as a function $h : V \cup E \rightarrow C$ such that

1. for every vertex $v \in V$ it holds that $h(v) = \textbf{vertex}(\kappa)$ and $\alpha(v) \in [\![\kappa]\!]$ for some content type κ and
2. for every edge $e \in E$ with $\varrho(e) = (v_1, v_2)$ it holds that $h(e) = \textbf{edge}(h(v_1), \kappa, h(v_2))$ and $\alpha(e) \in [\![\kappa]\!]$ for some content type κ.

We can then redefine strict conformance such that a property graph conforms to a graph-type core if there is a homomorphism from the property graph to the graph-type core.

To illustrate these semantics, consider the following graph-type core;

```
$personWithLongName = (:Person { name VARCHAR(50) })
$personWithShortName = (:Person { name VARCHAR(15) })
$degree = (:Degee { level VARCHAR(3), level VARCHAR(25)})
$hobby = (:Hobby { description VARCHAR(100) })
($personWithLongName-[:hasDegree]->(:$degree)
($personWithShortName-[:hasHobby]->(:$hobby)
```

Under the homomorphism-based semantics the following graph strictly conform the the preceding schema.

```
(v1 :Person { name="Mary Anne Cunningham" }
(v2 :Person { name="George Walsh" }
(v3 :Degree { level="MSc" topic="Philosophy"}
(v4 :Hobby { description="Bird watching" }
(v1) -[:hasDegree]-> (v3)
(v2) -[:hasHobby]-> (v4)
```

The following graph does not strictly confirm under homomorphism-based semantics:

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Degree { level="MSc" topic="Philosophy"}`
- `(v3 :Hobby { description="Bird watching" }`
- `(v1) -[:hasDegree]-> (v2)`
- `(v1) -[:hasHobby]-> (v3)`

It is possible to combine this approach with the ideas for dealing with dependent vertex types that were discussed in the previous section. This again allows us to give semantics to vertex types that only appear in edge types and not independently. For this we need a slightly more sophisticated notion of homomorphism.

Definition: Given a property graph a property graph $G = (V, E, \varrho, \alpha)$ and a graph-type core C we define a isolation-aware homomorphism from G to C as a function $h : (V \cup E) \rightarrow (C \cup C^{dep})$ where $C^{dep} = \{ \tau \mid \mathbf{edge}(\tau, \kappa, \sigma) \in C \vee \mathbf{edge}(\sigma, \kappa, \tau) \in C \}$ such that

1. for every vertex $v \in V$ that is not incident to any edge in G it holds that $h(v) \in C$,
2. for every vertex $v \in V$ it holds that $h(v) = \mathbf{vertex}(\kappa)$ and $\alpha(v) \in [\![\kappa]\!]$ for some content type κ and
3. for every edge $e \in E$ with $\varrho(e) = (v_1, v_2)$ it holds that $h(e) = \mathbf{edge}(h(v_1), \kappa, h(v_2))$ and $\alpha(e) \in [\![\kappa]\!]$ for some content type κ.

Subsequently we can define strict conformance such that a property graph strictly conforms to a graph-type core if there is an isolation-aware homomorphism from the property graph to the graph-type core.

To illustrate the difference with the previous semantics that is not isolation-aware consider the following adapted graph-type core:

- `$personWithLongName = :Person { name VARCHAR(50) }`
- `$personWithShortName = :Person { name VARCHAR(15) }`
- `$degree = (:Degee { level VARCHAR(3), level VARCHAR(25)})`
- `$hobby = (:Hobby { description VARCHAR(100) })`
- `($personWithLongName-[:hasDegree]->(:$degree)`
- `($personWithShortName-[:hasHobby]->(:$hobby)`

Note that `$personWithLongName` and `$personWithShortName` now define content types rather than vertex types. The following graph strictly conforms under isolation-aware homomorphism-based semantics:

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Person { name="George Walsh" }`
- `(v3 :Degree { level="MSc" topic="Philosophy"}`

- `(v4 :Hobby { description="Bird watching" }`
- `(v1) -[:hasDegree]-> (v3)`
- `(v2) -[:hasHobby]-> (v4)`

However, the following graph does not, since `v2` is now an isolated vertex for which there is no matching independent vertex type:

- `(v1 :Person { name="Mary Anne Cunningham" }`
- `(v2 :Person { name="George Walsh" }`
- `(v3 :Degree { level="MSc" topic="Philosophy"}`
- `(v1) -[:hasDegree]-> (v3)`

These changes in semantics have advantages and disadvantages:

- **Good:** The isolation-aware semantics gives a meaningful semantics for graph-type cores with vertex types that only occur in edge types.
- **Good:** The homomorphism semantics adds expressive power, although not as much as when the schema would actually be a graph giving abstract identity to vertex types.
- **Bad**: The added expressive power is quite subtle, and probably hard to understand by most users.
- **Bad**: Both have the disadvantage that, if distinct types can overlap, the computational complexity of validation becomes intractable, since it becomes similar to the general problem of checking for the existence of graph homomorphisms.

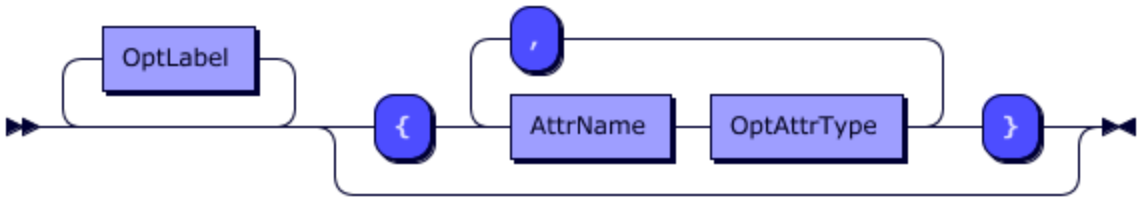
Graph-type cores with optional attributes

In this section we discuss adding optional attributes to graph-type cores and the semantics of this can be formalised.

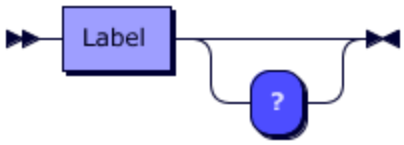
Concrete syntax

We allow in a content type declaration labels and attributes types to be annotated with “?” to indicate that the attribute is optional. For this purpose the syntax rule for content types is adapted as follows (new parts are highlighted in grey):

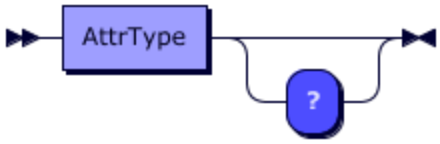
ContentType ::= OptLabel* ("{" AttrName OptAttrType ("," AttrName OptAttrType)* "}")?



OptLabel ::= Label ("?")?



OptAttrType ::= AttrType ("?")?



An example of a graph-type core with a single vertex type with optional attributes is:

```
(:City Place Capital? { name STRING, url URL? })
```

Vertices of this type might or might not have a label `Capital`, and they might or might not have an attribute with name `url`, but if they do then it must have a value of type `STRING`. So the following vertices are all **exact matches** of the previous type:

```
(v1 :City Place { name="The Hague" })
(v2 :City Place Capital { name="Amsterdam" })
(v3 :City Place { name="Manchester", url=https://www.manchester.gov.uk/ })
(v4 :City Place Capital { name="London", url=https://www.london.gov.uk/ })
```

The following are **over matches** of the previous type:

```
(v5 :City Place { name="The Hague", population=540,000 })
```

The following is **not a match** at all:

```
(v6 :City Place { name="The Hague", url=540,000 })
```

Abstract syntax

We define the notion of marked content type to incorporate the marking that some attributes are optional.

Definition: A marked content type is a pair (κ, M) where κ is a content type and M a subset of $\mathbf{dom}(\kappa)$ that indicates which attributes are not optional (so mandatory).

We extend the notion of graph-type core in a similar fashion:

Definition: A marked graph-type core is defined as a set of vertex types where a marked element type is defined as either a marked vertex type, denoted as **vertex**(m) where m is an marked content type, or a marked edge type, denoted as **edge**(σ , m, τ) where σ and τ are marked vertex types and m is a marked content type.

Formal semantics

We adapt the notion of matching for records and marked content types as follows:

Definition: Given a marked content type $m = (\kappa, M)$ we define the set of all over matches of m, denoted as $[\![m]\!]^*$, is defined as all finite partial functions $r : (\mathcal{A} \rightharpoonup \mathcal{V})$ such that (1) for every $(a \mapsto \sigma) \in \kappa$ and $(a \mapsto w) \in r$ it holds that $w \in [\![\sigma]\!]^*$ and (2) for every $(a \mapsto \sigma) \in \kappa$ such that $a \in M$ there is a pair $(a \mapsto w) \in r$ for some attribute value w. The set of all exact matches of m, denoted as $[\![m]\!]$, is defined as all $r \in [\![m]\!]^*$ such that for every pair $(a \mapsto w) \in r$ there is a pair $(a \mapsto \sigma) \in \kappa$ such that $w \in [\![\sigma]\!]$.

Note that the notion of over matching has now two requirements, namely (1) that every attribute in the record that is also mentioned in the content type must have a value of the specified type and (2) that all attributes that are not marked as optional indeed exist in the record.

For example, consider the marked content type ({ street ↦ **string**, city ↦ **string** }, { street }) , so a content type { street ↦ **string**, city ↦ **string** } where city is marked as optional, then

- the record { street ↦ *"Malet st"*, city ↦ *"London"* } is an over match and also an exact match,
- the record { street ↦ *"Malet st"*, city ↦ *"London"*, country ↦ *"UK"* } is an over match but not an exact match,
- the record { street ↦ *"Malet st"* } is an over match and an exact match,
- the record { city ↦ *"London"* } is neither an over match nor an exact match, and
- the record { street ↦ *5*, city ↦ *"London"*, country ↦ *"UK"* } is neither an over match nor an exact match, and
- the record { street ↦ *"Malet st"*, city ↦ *5*, country ↦ *"UK"* } is neither an over match nor an exact match.

The notions of subtyping and intersection types, which was already earlier defined for content types, can be generalized for marked content types as follows If we have two content types (κ_1, M_1) and (κ_2, M_2) we say that (κ_1, M_1) is a subtype of (κ_2, M_2), denoted as $(\kappa_1, M_1) \leq (\kappa_2, M_2)$, if $\kappa_1 \leq \kappa_2$ and $M_1 \supseteq M_2$. The implied intersection type can be formulated as follows. If we have two marked content types (κ_1, M_1) and (κ_2, M_2) then $(\kappa_1, M_1) \wedge (\kappa_2, M_2)$ denotes the marked content type $(\kappa_1 \wedge \kappa_2, M_1 \cup M_2)$. As before, we can observe again that this defines the greatest common subtype of the two types.

Discussion

Under strict conformance semantics an equivalent way of describing the semantics of optional attributes is to replace them with a set of simple types that describe all possible allowed patterns. For this we define the notion of *generated content types* for a certain marked content type $m = (\kappa, M)$, denoted as **gct**(m), and which is defined such that **gct**(m) = { { $(a \mapsto \tau) \mid (a \mapsto \tau) \in \kappa, a \in M \cup A$ } | $A \subseteq$ **dom**(κ) \ M }. We now transform a marked graph-type core C to a graph-type core C' where:

1. the simple vertex types in C' are { **vertex**(κ') | **vertex**(m) $\in C$, κ' $\in$ **gct**(m) } and
2. the vertex types in C' are { **edge**(σ, κ, τ) | **edge**(m_1, m, m_2) $\in C$, $\sigma \in$ **gct**(m_1), $\kappa \in$ **gct**(m), $\tau \in$ **gct**(m_2) }.

Example
Consider the following vertex type:

```
(:City Place Capital? { name STRING, url URL? })
```

Formally this is **vertex**(m) with $m = (\kappa, M)$ such that

- κ = { City ↦ **label**, Place ↦ **label**, Capital ↦ **label**, name ↦ **string**, URL ↦ **URL**}
- M = {City, Place, name}.

Then it holds for m that:
gct(m) = {
{ City ↦ **label**, Place ↦ **label**, name ↦ **string**},
{ City ↦ **label**, Place ↦ **label**, Capital ↦ **label**, name ↦ **string**},
{ City ↦ **label**, Place ↦ **label**, name ↦ string, URL ↦ **URL**},
{ City ↦ **label**, Place ↦ **label**, Capital ↦ **label**, name ↦ **string**, URL ↦ **URL**},
}

When we map these content types to vertex types, we get:

```
(:City Place { name STRING })
(:City Place Capital { name STRING })
(:City Place { name STRING, url URL })
(:City Place Capital { name STRING, url URL })
```

End example

Unfortunately describing the semantics of optional attributes this way does not work for the semantics of graph-type cores based on conformance and combinatorial conformance. This is illustrated by the following examples.

Begin example

For conformance-based semantics, consider the following graph-type core:

```
(:City Place { name STRING, url URL? })
```

After rewriting this becomes:

```
(:City Place { name STRING, url URL })
(:City Place { name STRING })
```

However, under conformance-based semantics this graph-type core is equivalent with:

```
(:City Place { name STRING })
```

This graph-type core is not equivalent with the original since it does not require the url property to be of type URL.

For combinatorial semantics consider the following graph-type core:

```
(:Manager { name STRING, birthdate STRING? })
(:Engineer { name STRING, birthdate INTEGER? })
```

After rewriting, this becomes:

```
(:Manager { name STRING, birthdate STRING })
(:Manager { name STRING })
(:Engineer { name STRING, birthdate INTEGER })
(:Engineer { name STRING })
```

This graph-type core, however, allows the following graph, which is not allowed by the original graph-type core:

```
(:Manager Engineer{ name=”Theobald” , birthdate=19801207 })
```

End example

Another reason to prefer a direct semantics for element types with optional attributes are the semantics of graph constraints. Consider for example the following type declaration:

```
$city = (:City Place Capital? { name STRING, url URL? })
```

And assume the following key constraint that refers to this type:

```
WHERE (c : $city) REQUIRE c.name IDENTIFIES c
```

Here `(c : $city)` denotes a graph matching pattern consisting of a single vertex pattern where `c` is a variable that can match any vertex of type `city`. It would not be correct to replace this constraint with a set of similar constraints where in each the type `city` is replaced with a different one of the vertex types generated by the type `city`. A similar observation can be made for cardinality constraints. It seems therefore preferable to describe directly what it means to belong to the type `$city`.

Yet another reason to give a direct semantics for element types with optional attributes is that it makes it harder to determine the computational tractability of schema validation. Rewriting such a type to the generated simple types can exponentially blow up the schema, which will lead to a very inefficient implementation when naively implemented, whereas the direct semantics leads more easily to an efficient implementation.

LDBC PGS-DM:DM-01

Meta-properties

LDBC—Property Graph Schema Working Group

Editor: Bei Li
Authors (alphabetical):
Alastair Green (alastair.j.green@gmail.com, Birkbeck, University of London),
Bei Li (bei@google.com, Google),
Borislav Iordanov (borislav.iordanov@gmail.com, HypergraphDB, Kobrix Software),
Dominik Tomaszuk (dtomaszuk@ii.uwb.edu.pl, University of Bialystok),
Jan Hidders (jan@dcs.bbk.ac.uk, Birkbeck, University of London),
Joshua Shinavier (joshsh@uber.com, TinkerPop, Uber),
Keith W. Hare (keith@jcc.com, JCC Consulting)

References

[GQL:DATA MODEL]	ISO/IEC JTC1 SC32 WG3 SXM-030r3 “The GQL property graph data model ”
[GQL:EWD V4]	ISO/IEC JTC1 SC32 WG3 WG3:MMX-010r2 “GQL Early Working Draft V4.2”
[GQL:DDL]	ISO/IEC JTC1 SC32 WG3 WG3:MMX-028r2 “Graph and graph type DDL”
[LDBC PGS-B:BAS-01r1]	ISO/IEC JTC1 SC32 WG3 WG3:MMX-069 “GS-Basic Overview report”
[G-CORE]	Angles, Renzo, et al. "G-CORE: A core for future graph query languages." Proceedings of the 2018 International Conference on Management of Data. 2018
[WIKIDATA:QUALIFIER]	https://www.wikidata.org/wiki/Help:Qualifiers
[WIKIDATA:STATEMENT]	https://www.wikidata.org/wiki/Help:Statements
[WIKI:VALID_TIME]	https://en.wikipedia.org/wiki/Valid_time
[WIKI:GQL_PATTERN]	https://en.wikipedia.org/wiki/GQL_Graph_Query_Language#Querying_with_visual_path_patterns
[APG]	Shinavier, Joshua, and Ryan Wisnesky. "Algebraic Property Graphs." arXiv preprint arXiv:1909.04881 (2019)

[ANNOTATED RDF]	Udrea, Octavian, Diego Reforgiato Recupero, and V. S. Subrahmanian. "Annotated rdf." ACM Transactions on Computational Logic (TOCL) 11.2 (2010): 1-41.
[TEMPORAL RDF]	Gutierrez, Claudio, Carlos Hurtado, and Alejandro Vaisman. "Temporal rdf." European Semantic Web Conference. Springer, Berlin, Heidelberg, 2005.
[LDBC PGS:AG-04r1]	Green, Alastair. LDBC PGS:AG-04r1 GS-Data Model attribute types[, date].
[LDBC PGS:AG-09r1]	Green, Alastair. LDBC PGS:AG09r1 GS-Datamodel--Examples of schema/path syntax for "metaproperties" options[, date].

Executive Summary

We motivate and discuss the syntactic representation and semantic interpretation of meta-properties in Property Graphs. Our proposal adds to the PG data model, and necessitates changes to the query language for matches, inserts and updates.

Meta-properties are motivated by common needs of annotating or qualifying assertions, which often are encoded on Property Graph properties, for example associating a valid time range during which the properties annotated are considered true in the real word.

This discussion paper proposes the desired CRUD behavior in the language layer, and two implementation approaches: (a) record type and (b) algebraic property graphs (APG). Both are compatible with the existing Property Graph definition. The option (a) extends the property type with a record type that intuitively represents a tree data structure, where nodes represent property values, and their descendants represent meta-properties. The option (b) extends the graph elements definition by including properties, thus properties can naturally be associated with properties.

An example user journey is crafted to demonstrate two approaches, with tentative syntax to better convey the idea.

Introduction

The definition of Property Graph data model in [GQL:DATA MODEL] which can be phrased as follows as in [LDBC PGS-B:BAS-01r1]:

> A property graph is defined as consisting of (1) a set of vertices such that each vertex has

> some associated vertex content, (2) a set of edges such that each edge has an associated (a) tail vertex, (b) edge content and (c) head vertex. The tail and head vertices of each edge must be in the set of vertices, and the content that is associated with vertices and edges is a finite record that maps attribute names to attribute values. Both vertices and edges are assumed to be represented by an abstract identity, and so it is possible that a graph contains two vertices with the same content, and two edges that have the same tail vertex, head vertex and content.

The formal definition of Property Graph we use in this document is as follows:

> **Definition:** A property graph is a tuple $G = (V, E, A, \varrho, \alpha)$ where
>
> - V is a set of vertex identities,
> - E is a set of edge identities,
> - A is a set of attribute identities such that V, E and A are pairwise disjoint,
> - $\varrho : E \rightarrow (V \times V)$ maps each edge identity to a pair of vertex identities
> - $\alpha : A \rightarrow ((V \cup E) \times \mathcal{A} \times \mathcal{V})$ maps each attribute identity to a triple containing (1) the element that it is an attribute of, (2) its name and (3) its value. Every attribute identity in A is identified by the first two components.

Specifically, In [GQL:EWD V4], the description related to property[1] in Property Graph can be summarized as: A graph has a set of zero or more nodes or edges. Each node or edge comprises zero or more properties. For each property, there is a name which is an identifier that is unique within the node or edge, and a value which can be any property value type. The property value type is defined as “the super type of all property values of any graph element of GQL-data.” in [GQL:EWD V4].

Furthermore, the [GQL:DATA MODEL] explains:

> ... the definition of a GQL graph data model proposed below is equivalent to the union of the data models of the following languages: Cypher, PGQL, SQL/PGQ, GSQL, and Tinkerpop/Gremlin. Put another way, these data models are all restricted editions of the GQL data model proposed.
>
> We are aware of two exceptions to this statement by Tinkerpop which allows for
> 1. properties on properties, and
> 2. multiple properties of the same name and type on an element.
>
> We believe that the first case could be most easily handled by allowing nesting of records, and the second by the use of collections as or within property values. Alternatively, these

[1] Note in this document, the term ‘attribute’ and ‘property’ are used interchangeably.

cases could be expressed using graph structures to represent trees rooted in an element. We would welcome other opinions regarding such features.

This paper we primarily focus on addressing the issues in the first exception above, namely “properties on properties” which we call “meta-properties” in this document, with the motivation described in the section below.

The authors agree with the rendition of multiple property values as a named collection whose elements are of a specified type. The rest of this paper does not address that point further.

Motivating use cases

Some areas where meta-properties can be useful are:

- **Provenance**. Systems that operate over data collected from a large number of unvetted sources must track property value provenance in order to justify or rank the contribution of independent sources.
- **Fuzzy** and **trust**. The fuzzy and trust areas, although having different meanings, can be represented in a similar manner e.g. the annotation as a value between 0 and 1.
- **Valid time**. Any value of property can be valid over different periods of time. That can be represented as a set of disjoint time intervals that represent each a period of time when the property value was valid.
- **Timestamp** for the last update (This is slightly different than full temporal support)
- **Units** (feet, meters, pounds, kilograms, etc.)
- **Location** (Latitude, Longitude and associated coordinate reference system) associated with information?

Here we expand two of the use cases below to further illustrate the motivation.

Wikidata qualifiers

The core concept of the Wikidata data model [WIKIDATA:STATEMENT], [WIKIDATA:QUALIFIER] can be described as the following:

- Statements are used for recording data about an item
- Statements consist of (at least) one property-value pair
- Statements can be **further annotated or contextualized** with additional values, as well as optional qualifiers, references, and ranks

In Wikidata, qualifiers are used extensively. The example below from [WIKIDATA:QUALIFIER] shows two measurements of the population of Berlin, at two points in time, with two different estimation processes modelled using qualifiers.

```
Berlin (item)
   population (property) → 3,500,000 (property value)
      point in time (qualifier) → 2005 (qualifier value)
      determination method (qualifier) → estimation process (qualifier value)
   population (property) → 3,440,441 (property value)
      point in time (qualifier) → 2010 (qualifier value)
      determination method (qualifier) → census (qualifier value)
```

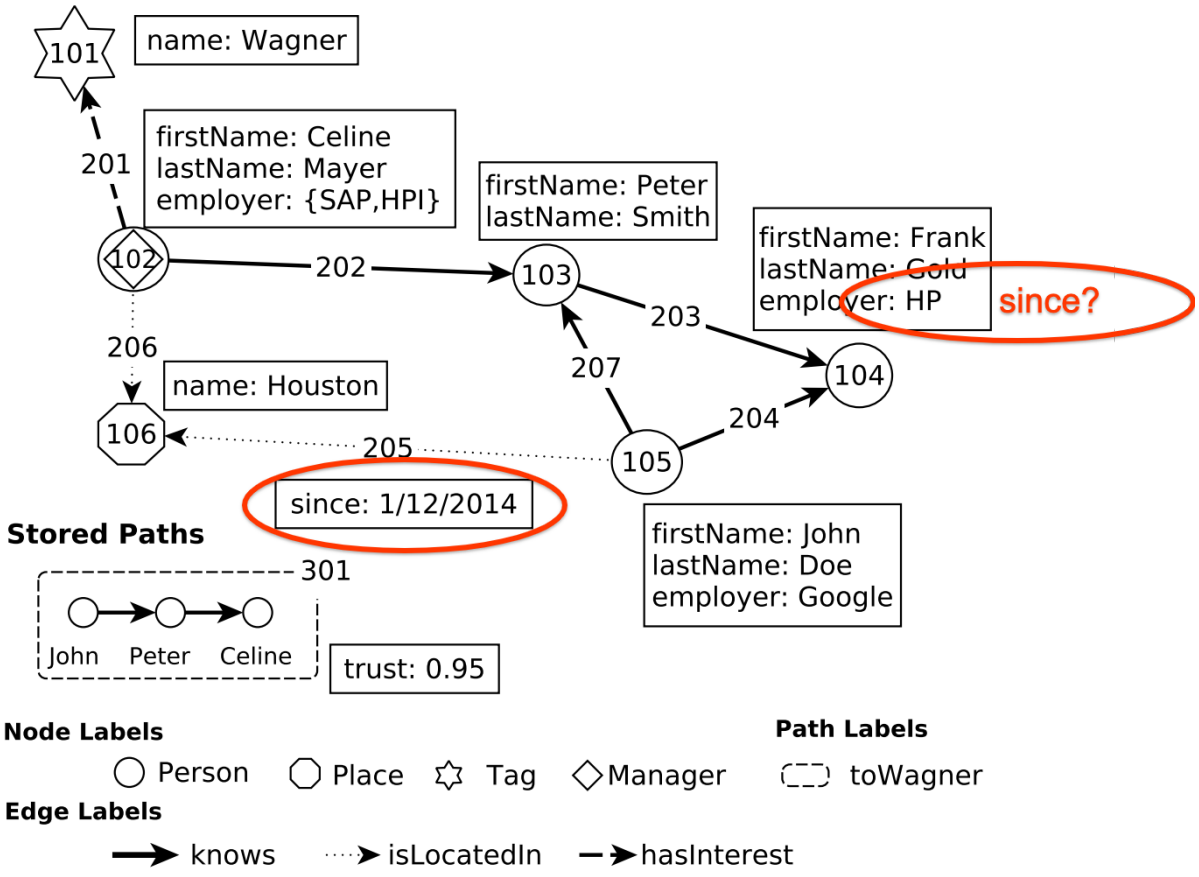
Mapping to the GQL Property Graph data model, statements can be represented as either edges or node properties. For example, the two statements (a) “Berlin is the capital of Germany” and (b) “Berlin has a population of 3.5 million”, using GQL pattern syntax (see [WIKI:GQL_PATTERN] for an overview, [GQL:EWD V4] for the details), can be represented as (Berlin) -[:CAPITAL_OF]-> (Germany), and (Berlin { population: 3.5 million }) repectively.

For edges, qualifiers can be modelled using edge properties. For example, to add a qualifier “since 1871” to the statement “Berlin is the capital of Germany”, one can do the following: (Berlin) -[:CAPITAL_OF **{ since: 1871 }**]-> (Germany). But for node properties, there is no native way of representing qualifiers; ‘population’ is already a property and properties on properties is not yet supported in GQL Property Graph.

Valid time

Valid time [WIKI:VALID_TIME] is used to represent the time period during which the assertions are considered true. It can naturally be represented as properties on edges, indicating the time period when the edge assertion is considered true. See the diagram below from [G-CORE], the edge labelled 205 has a property indicating its starting time. Using GQL pattern syntax, the assertion can be encoded as (105) -[205 **{ since: 2014-1-12 }**]-> (106).

However, the same could be applied to properties. For example, for the node labeled 104, the employer property could also be qualified with a ‘since’ value. Since the ‘employer’ is already a property on the node, further qualifications of it can not be represented natively in GQL Property Graph yet.



Source: [G-CORE]

Requirements

From the user cases above, we have the following requirements distilled:

Must-haves

1. Any extensions to the PG data model should not break existing work.
2. Support for meta-properties of simple or complex-valued properties.
3. Meta-properties should be queryable similar to regular properties.
4. The typing system should be able to control which metaproperties are allowed where and what their values are. This applies both at the graph schema level and within the property values.

Nice-to-haves

5. Support for meta-properties of components of complex values (e.g. record attributes, list elements, map keys or map values). For example, an Address property which is broken down to street number, name and city could have meta-properties at the level of the Address record as a whole, but also at the level of each constituent components.

Overview of the proposals

The highlevel ideas are:

- Conceptually properties and their meta-properties can be viewed as a tree structure.
- The structure has the shape of { property name: value { meta-property name: value } } where the component { meta-property name: value } annotates the property “property name: value”.
- The depth of meta-properties is arbitrary, i.e., meta-properties can also attach to meta-properties. For example, such a structure is allowed: { property name: value { meta-property name_1: value { meta-property name_2: value } } } where the component {meta-property name_2: value } annotates the meta-property “meta-property name_1: value”. The depth is implementation-dependent.
- Meta-properties can be accessed similarly to regular properties, and can be used in CRUD operations.

There are two perspectives of the tree structure representation:

- Perspective 1 uses a complex data type to represent the tree structure, conceptually similar to JSON or XML documents. With this, meta-properties can be simply handled by an addition to the value type system.
- Perspective 2 views properties as a first-class graph element with their own identities. Just like other graph elements such as nodes and edges, properties can attach to properties, thus fulfill the meta-property requirements.

Though the two perspectives represent distinct trains of thought, with different origins, which lead to a set of behaviors which is largely or wholly the same. To the extent that these two perspectives have been discussed as “alternatives” it has been perceived that the behaviors manifested by perspective 1 subsume those of perspective 2. We will examine whether this is actually the case in more detail later.

Detailed description of the proposals

In this section we first describe the meta-properties from the query language’s perspective, including Data Definition Language (DDL), Data Query Language (DQL) and Data Manipulation Language (DML). Note the language syntaxes proposed are for illustration only, and open for discussion. In addition, the DDL syntax used is conceptually aligned to the [GQL:DDL] though does not necessarily strictly conform to it.

We then focus on the formal definitions of meta-properties which hopefully will also shed light to implementation choices.

Query language

Simple-valued properties

DDL

It's possible to attach meta-properties on node or edge properties with simple values.

In the example below, the node property 'population' with the INTEGER type can have a meta-property named 'point_in_time' with the type 'DateTime'.

```
CREATE GRAPH TYPE Stats {
  (City :City { name STRING, population INTEGER { point_in_time DateTime } }
}
```

It's also possible to attach meta-properties to meta-properties. In the example below, the 'confidence_score' meta-property is attached to the meta-property 'point_in_time'.

```
CREATE GRAPH TYPE Stats {
  (City :City
        { name STRING,
          population INTEGER { point_in_time DateTime { confidence_score FLOAT } }
        })
}
```

Meta-properties on edge properties behave exactly the same as meta-properties on node properties. In the example below, for the edge type 'ContainedBy', there is a meta-property 'provenance' for the edge property 'since'.

```
CREATE GRAPH TYPE Stats {
  (City  :City  { name STRING })
  (State :State { name STRING })

  (City)-[ContainedBy :CONTAINED_BY
                      { since INTEGER { provenance STRING } } ]->(State)
}
```

The rest of the document will skip edge meta-properties for simplicity.

DML

Insert

It's possible to insert a property and its meta-properties in the same statement. The same applies to meta-properties on meta-properties. In the example below, the node property 'population', the meta-properties 'point_in_time' and 'confidence_score' are inserted in the same statement.

Input graph:

```
// empty
```

Query:

```
INSERT (n:City
        {
          name: "Berlin",
          population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
        })
RETURN n
```

Resulting graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Update

A meta-property can be updated together with the regular property it's attached to:

Input graph:

```
(:City {
   name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
UPDATE (n:City { name = "Berlin", population = 3.5M })
SET n.population = 3.5M { point_in_time: 2005 { confidence_score: 1.0 } }
```

One can also use the dot notation to directly update meta-properties:

Query:

```
UPDATE (n:City { name = "Berlin", population = 3.5M })
SET n.population.point_in_time.confidence_score = 1.0
```

Resulting graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 1.0 } }
 })
```

Meta-property variables

A meta-property can also be updated using meta-property variables. In examples below, a variable is associated with the property ‘point_in_time’ and the update operations can be directly applied to the variable.

Input graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
// Update the meta-property 'point_in_time' by the meta-property variable $mp.
UPDATE (:City { name = "Berlin", population { mp:point_in_time } })
SET mp = 2005-5 { confidence_score: 0.99 }
```

Resulting graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005-5 { confidence_score: 0.99 } }
 })
```

Delete

When deleting a property, all meta-properties attached to it will be deleted.

Input graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
MATCH (n:City { name = "Berlin" })
REMOVE n.population
```

Resulting graph:

```
(:City { name: "Berlin" })
```

Meta-property variables

You can also delete a meta-property using a meta-property variable:

Input graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
// Delete the meta-property 'point_in_time' using the meta-property variable $mp.
MATCH (:City { name = "Berlin", population { mp:point_in_time } })
REMOVE mp
```

Resulting graph:

```
(:City { name: "Berlin", population: 3.5M })
```

Since a meta-property variable can be associated with a meta-property at any depth, you can also do the following:

Input graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
// Delete the meta-property 'confidence_score' using the
// meta-property variable $c.
MATCH (:City { name = "Berlin",
               population { point_in_time { c:confidence_score } }
             })
REMOVE c
```

Resulting graph:

```
(:City { name: "Berlin", population: 3.5M { point_in_time: 2005 } })
```

DQL

Read

When reading a property, regardless if there are meta-properties attached to it, only the property value will be returned. The same applies to a meta-property; reading it should just return the value, not the meta-properties attached to it. One of the motivations is to keep the behaviors backward compatible.

In the example below, though it has a meta-property ‘point_in_time’, reading property ‘population’ will only return the population value, without its meta-properties. The meta-properties need to be explicitly requested. Dot notation is tentatively used to navigate to the meta-properties from properties in this document.

Input graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
MATCH (n:City { name = "Berlin" })
RETURN n.population, n.population.point_in_time
```

Binding table:

population	population.point_in_time
3.5M	2005

Options are provided if the qualified values are desired. In the example below, we use the tentative syntax of property / meta-property name followed by “.*” to indicate that the qualified values are desired.

Input graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
// Read the qualified value of 'population'
MATCH (n:City { name = "Berlin" })
RETURN n.population.*
```

Binding table:

population.*
3.5M { point_in_time: 2005 { confidence_score: 0.99 } }

Filtering

A meta-property can be used in filtering conditions (predicates) just like a regular property, as illustrated below.

Comparison

A meta-property can be used in comparison.

Input graph:

```
(:City
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
MATCH (n:City { population.point_in_time = 2005 })
RETURN n.name

// Equivalent query using WHERE clause.
MATCH (n:City)
WHERE n.population.point_in_time = 2005
RETURN n.name
```

Binding table:

name
“Berlin”

Existence

A meta-property can be used in existence tests for filtering.

Input graph:

```
(:City
```

```
 { name: "Berlin",
   population: 3.5M { point_in_time: 2005 { confidence_score: 0.99 } }
 })
```

Query:

```
// Match nodes that have a property named 'population' with
// a meta-property named 'point_in_time'.
MATCH (n:City { population { point_in_time } })
RETURN n.name
```

Binding table:

name
“Berlin”

Collection-valued properties

For a collection-valued property, meta-properties can be associated to the property, or individual elements in the collection.

DDL

In the example below, the collection-valued property “zip_code” has two types of meta-properties: (1) the meta-property “point_in_time” on the property level, (2) the meta-properties “creation_tine” for the individual elements.

```
CREATE GRAPH TYPE Stats {
  (City :City
        { name STRING,
          zip_codes [INTEGER { creation_time DateTime }] { point_in_time DateTime }
        })
}
```

DML

Insert

When creating a collection-valued property, both meta-properties on the property level and individual element level can be set.

Input graph:

```
// empty
```

Query:

```
INSERT (n:City
        {
          name: "Berlin",
          zip_codes: [10115 { creation_time: 1962 },
                      10117 { creation_time: 1965 }] { point_in_time: 2005 }
        })
RETURN n
```

Resulting graph:

```
(:City
 { name "Berlin",
   zip_codes: [10115 { creation_time: 1962 },
               10117 { creation_time: 1965 }] { point_in_time: 2005 }
 })
```

Update

Updating property level meta-properties can be done in the same way as updating meta-properties for simple-valued properties as explained above.

Updating elements level meta-properties requires updating the entire collection-typed property:

Input graph:

```
(:City
 { name "Berlin",
   zip_codes: [10115 { creation_time: 1962 },
               10117 { creation_time: 1965 }] { point_in_time: 2005 }
 })
```

Query:

```
UPDATE (n:City { name = "Berlin" })
SET n.zip_codes = [10115 { creation_time: 1963 },
                   10117 { creation_time: 1966 }] { point_in_time: 2005 }
```

Resulting graph:

```
(:City
 { name: "Berlin",
   zip_codes: [10115 { creation_time: 1963 },
               10117 { creation_time: 1966 }] { point_in_time: 2005 }
 })
```

Delete

Deleting property level meta-properties can be done in the same way as deleting meta-properties for simple-valued properties as explained previously.

Deleting elements level meta-properties requires updating the entire property as illustrated below.

Input graph:

```
(:City { name: "Berlin",
         zip_codes: [10115 { creation_time: 1962 },
                     10117 { creation_time: 1965 }] { point_in_time: 2005 } })
```

Query:

```
UPDATE (n:City { name = "Berlin" })
SET n.zip_codes = [ 10115, 10117 ] { point_in_time: 2005 }
```

Resulting graph:

```
(:City { name: "Berlin", zip_codes: [ 10115, 10117 ] { point_in_time: 2005 } })
```

DQL

Read

Reading a collection-type property with meta-properties on elements will only return the collection values without the meta-properties on the elements, as illustrated below.

Input graph:

```
(:City
 { name "Berlin",
   zip_codes: [10115 { creation_time: 1962 },
               10117 { creation_time: 1965 }] { point_in_time: 2005 }
 })
```

Query:

```
MATCH (n:City { name = "Berlin" })
RETURN n.zip_codes, n.zip_codes.point_in_time
```

Binding table:

zip_codes	zip_codes.point_in_time

[10115, 10117]	2005

To read meta-properties on elements, ".*" operator is needed:

Query:

```
MATCH (n:City { name = "Berlin" })
RETURN n.zip_codes.*
```

Binding table:

zip_codes.*
[10115 { creation_time: 1962 }, 10117 { creation_time: 1965 }] { point_in_time: 2005 }

Filtering

Property level meta-properties can be used in filtering conditions.

Note in this proposal we defer the discussion of filtering using element level meta-properties.

Input graph:

```
(:City
 { name "Berlin",
   zip_codes: [10115 { creation_time: 1962 },
               10117 { creation_time: 1965 }] { point_in_time: 2005 }
 })
```

Query:

```
MATCH (n:City { zip_codes.point_in_time = 2005 })
RETURN n.name

// Equivalent query.
MATCH (n:City)
WHERE n.zip_codes.point_in_time = 2005
RETURN n.name
```

Record-valued properties

DDL

It's possible to attach meta-properties to both the property level and components level of a record-valued property. In the example below, the 'since' property is attached to the 'mayor' property level, while the two 'last_edit' meta-properties are attached to the components 'first_name', and 'last_name' respectively.

```
CREATE GRAPH TYPE Stats {
  (City :City
        { name STRING,
          mayor {
            first_name STRING { last_edit UINT64 },
            last_name  STRING { last_edit UINT64 }
          } { since DateTime }
        })
}
```

DML

Insert

It's possible to insert a record-valued property, with meta-properties at components level and property level specified.

Input graph:

```
// empty
```

Query:

```
INSERT (n:City {
          name: "Berlin",
          mayor: {
            first_name: "Michael" { last_edit: 1420070400 } ,
            last_name:  "Müller"  { last_edit: 1420070400 } } {
              since: 2014-12-11 }
            }
          })
RETURN n
```

Resulting graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 } ,
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12-11 }
 })
```

Update

A meta-property can be updated together with the property it's attached to:

Input graph:

```
(:City { name: "Berlin" })
```

Query:

```
UPDATE (n:City { name = "Berlin" })
SET n.mayor = {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12-11 }
```

Resulting graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12-11 }
 })
```

Meta-property variables

A meta-property can also be updated by using meta-property variables. In examples below, a variable is associated with the property ‘mayor’ and the updating operations are directly applied on the variable. The operations can also be directly applied to the variable ‘ms’ for the meta-property ‘since’.

Input graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12-11 }
 })
```

Query:

```
UPDATE (:City { name = "Berlin", mayor { ms:since } })
SET ms = 2014-12
```

Resulting graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
```

```
    last_name:  "Müller"  { last_edit: 1420070400 }
  } { since: 2014-12 }
 })
```

Delete

When deleting a property, all meta-properties attached to it will also be deleted.

Input graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12 }
 })
```

Query:

```
MATCH (n:City { name = "Berlin" })
REMOVE n.mayor
```

Resulting graph:

```
(:City { name: "Berlin" })
```

You can also delete a meta-property directly:

Input graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12 }
 })
```

Query:

```
MATCH (:City { name = "Berlin" })
REMOVE m.mayor.since
```

Resulting graph:

```
(:City {
```

```
  name: "Berlin",
  mayor: {
    first_name: "Michael" { last_edit: 1420070400 },
    last_name:  "Müller"  { last_edit: 1420070400 }
  }
})
```

Meta-property variables

You can also delete a meta-property directly using a meta-property variable:

Input graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12 }
 })
```

Query:

```
MATCH (:City { name = "Berlin", mayor { ms:since } })
REMOVE ms
```

Resulting graph:

```
(:City {
  name: "Berlin",
  mayor: {
    first_name: "Michael" { last_edit: 1420070400 },
    last_name:  "Müller"  { last_edit: 1420070400 }
  }
})
```

DQL

Read

It's possible to read record-valued properties, without any meta-properties associated.

Input graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
```

```
    last_name:  "Müller"  { last_edit: 1420070400 }
  } { since: 2014-12-11 }
 })
```

Query:

```
MATCH (n:City { name = "Berlin" })
RETURN n.name, n.mayor
```

Binding table:

name	mayor
“Berlin”	{ first_name: "Michael", last_name: "Müller" }

It’s also possible to read record-valued properties together with their meta-properties using ‘.*’.

Input graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12-11 }
 })
```

Query:

```
MATCH (n:City { name = "Berlin" })
RETURN n.name, n.mayor.*
```

Binding table:

name	mayor.*
“Berlin”	{ first_name: "Michael" { last_edit: 1420070400 }, last_name: "Müller" { last_edit: 1420070400 } } { since: 2014-12-11 }

It’s possible to read specific components of record-valued properties without meta-properties associated.

Input graph:

```
(:City
```

```
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12-11 }
 })
```

Query:

```
MATCH (n:City { name = "Berlin" })
RETURN n.name, n.mayor.first_name
```

Binding table:

name	mayor.first_name
“Berlin”	"Michael"

It’s also possible to read specific components of record-valued properties with their meta-properties.

Input graph:

```
(:City
 { name: "Berlin",
   mayor: {
     first_name: "Michael" { last_edit: 1420070400 },
     last_name:  "Müller"  { last_edit: 1420070400 }
   } { since: 2014-12-11 }
 })
```

Query:

```
MATCH (n:City { name = "Berlin" })
RETURN n.name, n.mayor.first_name.*
```

Binding table:

name	mayor.first_name.*
“Berlin”	"Michael" { last_edit: 1420070400 }

Disambiguation

Since meta-properties and record values have similar proposed behaviors, it’s possible to introduce ambiguities. For example, in the DDL below there is a node with a record-valued ‘score’ property. The record has a field called ‘confidence’. The property has a meta-property, which is also called ‘confidence’.

DDL

```
CREATE GRAPH TYPE Stats {
  (City :City
        { score {
            livability FLOAT,
            confidence FLOAT   // confidence about the city
          } {
            confidence FLOAT   // confidence about the `score`
          }})
}
```

Read

Since the record-valued ‘score’ property has a field ‘confidence’, and it also has a meta-property called ‘confidence’. Just by using “score.confidence”, it’s impossible to distinguish the two cases.

Input graph:

```
(:City {
  name: "Berlin",
  score: { value: 42, confidence: 1 } { confidence: 0.9 }
})
```

Query:

```
MATCH (n:City { name = "Berlin" })
// Which one the 'confidence' is referring to? The record field or the
meta-property?
RETURN n.score.confidence
```

To disambiguate, we propose to use ‘@’ to access meta-properties, “.” to access record field while “#” or “#*” to access both values and their meta-properties:

```
MATCH (n:City { name = "Berlin" })
RETURN n.score.confidence, n.score.*
```

Binding table:

score.confidence	score.*
1	{ value: 42, confidence: 1 }

Query:

```
MATCH (n:City { name = "Berlin" })
```

```
RETURN n.score@confidence, n.score@*
```

Binding table:

score@confidence	score@*
0.9	{ confidence: 0.9 }

Query:

```
MATCH (n:City { name = "Berlin" })
RETURN n.score#*, n.score#
```

Binding table:

score#*	score#

{ value: 42, confidence: 1 } {
  confidence: 0.9
}
	{ value: 42, confidence: 1 } {
	  confidence: 0.9
	}

Formal definitions

As mentioned previously, the meta-properties are informally viewed as a tree structure. There are two perspectives regarding how the tree structure is constructed. Here we present both and compare them.

Perspective 1: Record type

This proposal is based on the ideas that (a) for any property value, we can annotate it with a record describing meta-properties, and (b) for complex property values, we can do this at any level for any component of the values.

Examples

In the following example, ‘population’ value is annotated by the meta-property ‘point_in_time’ and ‘point_in_time’ can be further annotated by the meta-property ‘confidence_score’ as highlighted below:

```
CREATE GRAPH TYPE Stats {
  (City :City
        { name STRING,
          population INTEGER { point_in_time DateTime { confidence_score FLOAT } }
        })
```

```
}
```

In the following example, elements in ‘zip_code’ are annotated by the meta-property ‘creation_time’ as highlighted below:

```
CREATE GRAPH TYPE Stats {
  (City :City
        { name STRING, zip_codes [INTEGER { creation_time DateTime }] })
}
```

Formal definition

We generalize the set of property values $\mathcal{V}$ to the set of annotated property values $\mathcal{V}^{@}$ to include annotated values. This is done based on a generalisation of the given basic types and type constructors, as follows.

Definition: The set of **annotated attribute types** $\mathcal{T}_{\text{attr}}^{@}$ is recursively defined as a type of the form $type_1@type_2$ where $type_1$ is either a basic type, a record type with fields that have all annotated attribute types, or a collection type parameterized with annotated attribute types and $type_2$ is a record type where all field types are again of annotated attribute types.

Definition: The **semantics of an annotated attribute type** *type* in $\mathcal{T}_{\text{attr}}^{@}$ is denoted as $[\![type]\!]$ and consists of pairs of attribute values, denoted as $v@w$, and co-inductively (i.e., following the same induction as the definition of annotated attribute types) defined such that $v@w \in [\![type_1@type_2]\!]$ if and only if v is of type $type_1$ and w is of type $type_2$.

The proposal then simply consists of requiring annotated attribute types for all attributes in a schema. Note that this also accounts for unannotated values since these are described by an annotated attribute type of the form *type*@{}, i.e., a type with an empty annotation record.

Note that annotated attribute types can be arbitrarily nested, but cannot be of the form ((*type*@{...})@{...}). However the definition does explicitly allow indirect nesting of annotations and nesting in the record type describing the annotation. We do allow for example:

- **Integer**@{ confidence : **Real**@{} }
  An integer with a confidence annotation.
- **Integer**@{ confidence : **Real**@{ confidence-editor : **String**@{} } }
  An integer with a confidence annotation, which in turn is annotated with who edited that annotation.
- **Set**[**String**@{ confidence : **Real**@{} }]@{ set-editor : **String**@{} }
  A set of strings, where each string is annotated with confidence, and the set as a whole is annotated with who edited the set.

- **Set**[**String**@{ confidence : **Real**@{ confidence-editor : **String**@{} } }]@{ set-editor : **String**@{} }
  A set of strings, where each string is annotated with a confidence level, where this confidence is annotated with who edits it, and the set as a whole is annotated with who edits the set.

NOTE: Also here we could restrict the nesting depth of annotations within annotations. Since we do not do so, we can have properties of properties of properties, et cetera.

Examples

Type	Example values

`STRING`
 	
`STRING@{}`
	`“Berlin”`

`INTEGER@{ point_in_time: DateTime }`	`2.5M`
	
	`2.5M { point_in_time: 2005 }`

`Set[INTEGER {`
`  creation_time: DateTime`
`}]@{`
`  point_in_time: DateTime`
`}`
	`[10115 { creation_time: 1962 },`
	` 10117 { creation_time: 1965 }] {`
	`  point_in_time: 2005`
	`}`
	
	`[10115 { creation_time: 1962 },`
	` 10117 { creation_time: 1965 }]`
	
	`[10115 { creation_time: 1962 },`
	` 10117]`
	
	`[10115, 10117]`
	
	`null { point_in_time: 2005 }`

`{ first_name: STRING }@{`
`  since: DateTime`
`}`
	`{ first_name: “Michael” } {`
	`  since: 12/2004`
	`}`
	
	`{ first_name: “Michael” }`
	
	`null { since: 2004-12 }`

Observations

In this section we compare the Record Type approach against the aforementioned

requirements:

1. Any extensions to the PG data model should not break existing work.
   [Yes] The Record Type approach simply extends property value types.
2. Support for meta-properties of simple or complex-valued properties:
   [Yes] The property value types can support both types.
3. **[Yes]** Meta-properties should be indexable and queryable just like regular properties.
4. The typing system should be able to control which metaproperties are allowed where and what their values are. This applies both at the graph schema level and within the property values.
   [Yes] The type definition of the Record Type can control it.
5. Support for meta-properties of components of complex values (e.g. record attributes, list elements, map keys or map values). For example, an Address property which is broken down to street number, name and city could have meta-properties at the level of the Address record as a whole, but also at the level of each constituent components.
   [Yes] For complex types such as Address that have sub-components, meta-properties can be attached to them. In fact the meta-properties can be attached to any level of the nested structure.

Perspective 2: Algebraic Property Graphs

This proposal is based on the idea from [APG] that at the property graph level we lift the notion of property to the level of element, by which we mean here that (a) it is represented in the model by an abstract identity, like vertices and edges are, (so attributes that have the same name and value can be distinct) and (b) that it can have properties, like vertices and edges can. We do disallow that properties are directly or indirectly properties of themselves.

Examples

We will use the example graph below to clarify the idea:

```
(:City
 { name: "Berlin",
   population: 3.5M {
     point_in_time: 2005 { writer: "admin" },
     provenance: "wikipedia"
  }
 })
```

From the APG's perspective, the graph above is modelled with six graph elements: the #1 is the city node, the rest are the properties and meta-properties. Each graph element has an identity. The identity is represented as $variable for convenience.

The graph element #1 city has the identifier $n, and there are two properties attached to it which are $nn and $np, which is property 'name' and 'population' respectively. There are no properties

attached to the property $nn. For $np, there are two properties attached to it: $np1 and $np2. Furthermore, the property $np11 is attached to the property $np1.

```
1. $n                                          // node
2. $nn:   <$n    name           "Berlin">      // property
3. $np:   <$n    population     3.5M>          // property
4. $np1:  <$np   point_in_time  2005>          // meta-property
5. $np2:  <$np   provenance     "wikipedia">   // meta-property
6. $np11: <$np1  writer         "admin">       // meta-meta-property
```

Note to model the collection-typed properties, the APG approach has the option of using multi-valued properties, in addition to collection-typed values. With this option, meta-properties can be attached to each element.

Input graph:

```
(:City
 { name: "Berlin",
   population: [
     3.5M {
       point_in_time: 2005 { writer: "admin" },
       provenance: "wikipedia"
     },
     3.4M {
       point_in_time: 2010 { writer: "admin"}
       provenance: "wikipedia"
     }
 })
```

Which is modelled in AGP as:

```
1. $n                                                  // node
2. $nn:        <$n     name           "Berlin">      // property
3. $np1:       <$n     population     3.5M>          // property
4. $np11:      <$np1   point_in_time  2005>          // meta-property
5. $np12:      <$np1   provenance     "wikipedia">   // meta-property
6. $np111:     <$np11  writer         "admin">       // meta-meta-property
7. $np2:       <$n     population     3.4M>          // property
8. $np21:      <$np2   point_in_time  2010>          // meta-property
9. $np22:      <$np2   provenance     "wikipedia">   // meta-property
10. $np211:    <$np21  writer         "admin">       // meta-meta-property
```

Similar to the previous example, graph element #1 is the city node and #2 is the name of the city. Elements #3 - #6 are the first population property and its meta-properties, while elements #7 - #10 are for the second population property.

Note that in this example, there are no collection-typed values; they are modelled using multi-valued properties, thus is in bag semantics.

Formal definition

Definition: A property graph with meta-properties is a tuple $G = (V, E, A, \varrho, \alpha)$ where

- V is a set of vertex identities,
- E is a set of edge identities,
- A is a set of attribute identities such that V, E and A are pairwise disjoint,
- $\varrho : E \rightarrow (V \times V)$ maps each edge identity to a pair of vertex identities
- $\alpha : A \rightarrow ((V \cup E \cup A) \times \mathcal{A} \times \mathcal{V})$ maps each attribute identity to a triple containing (1) the element that it is an attribute of, (2) its name and (3) its value, so that an attribute cannot directly or indirectly be its own attribute.

For the property values we assume that they are typed under the type mechanism described earlier and so the schema will specify for each property a type from the set of attribute value types $\mathcal{T}_{\text{attr}}$.

NOTE: In this definition, there can be multiple properties with the same name attached to the same element, which essentially enables the multi-valued properties semantics. Furthermore, without restriction, it can even allow multiple properties with the same name and value attached to the same element. We view this as a generalization of the multigraph semantics.

Observations

In this section we compare the APG approach against the aforementioned requirements:

1. Any extensions to the PG data model should not break existing work:
   [Yes] The AGP approach extends the Property Graph element definition by including properties in addition to nodes and edges so that properties can also attach to properties in addition to nodes and edges.
2. Support for meta-properties of simple or complex-valued properties:
   [Yes] The property value types can support both types.
3. **[Yes]** Meta-properties should be indexable and queryable just like regular properties.
4. The typing system should be able to control which metaproperties are allowed where and what their values are. This applies both at the graph schema level and within the property values.
   [Yes] For each property type, the type of elements it can attach to can be defined.
5. Support for meta-properties of components of complex values (e.g. record attributes, list elements, map keys or map values). For example, an Address property which is broken down to street number, name and city could have meta-properties at the level of the Address record as a whole, but also at the level of each constituent components.

[Partially yes] For properties with complex types such as Address that have sub-components, properties can only be attached to the property level, not to the sub-components. However, for collection types, APG is able to model meta-properties on elements using multi-valued properties.

Summary

Main results and contributions

In this paper we have discussed the necessity of meta-properties in Property Graph. A few real-world use cases are presented to further clarify the importance of the construct. We have also put forward a preliminary proposal of the DDL and CRUD behaviors of meta-properties.

We have explained two perspectives of the formulations of meta-properties: record type and algebraic property graphs. The record type approach extends the Property Graph type system by including annotated attribute types. Besides solving the meta-properties problem in the paper, it has potential application in other domains. The algebraic property graphs approach extends the graph element definition by including properties so that properties can also be attached to properties, in addition to nodes and edges.

We have compared the two perspectives.

Observations

Here we summarize the behaviors of the two perspectives:

Requirements	Record type	APG
1, Any extensions to the PG data model should not break existing work.	Yes	Yes
2, Support for meta-properties of simple or complex values.	Yes	Yes
3, Meta-properties should be indexable and queryable just like regular properties.	Yes	Yes
4, The typing system should be able to control which metaproperties are allowed where and what their values are.	Yes	Yes
5, Support for meta-properties of components of complex values	Yes	Partially yes

Open questions

1. We have proposed a pseudocode notation for meta-properties, only for non-normative explanatory purposes. To put meta-properties into practice, a special syntax should be developed.
2. We do not refer to specific data types in this proposal. But we assume that annotations can be of the same data type as property values.

Conclusion

Expressivity and useability are fundamental to the adoption of a data model. By extending the Property Graph data model to include meta-properties, the proposals in this document enable a variety of new use cases, and have laid a solid foundation for future advances.

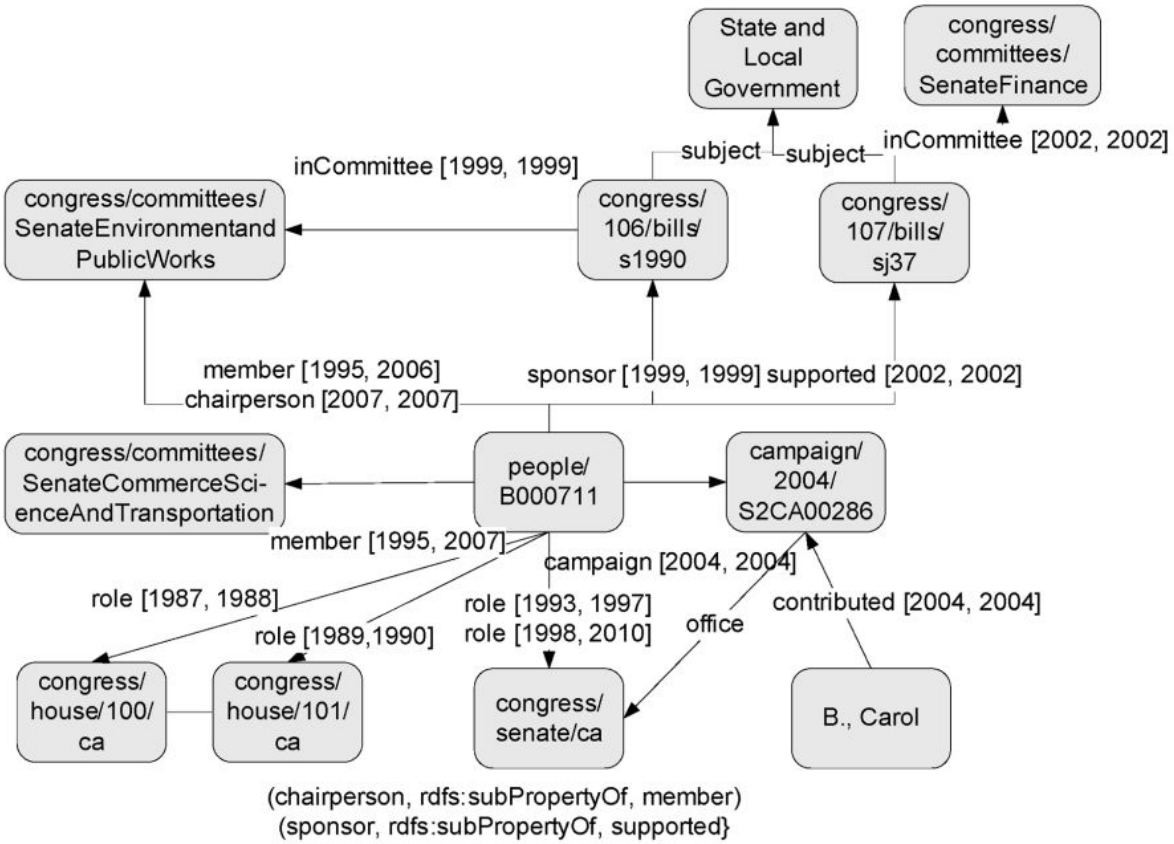
Appendix I

More use cases

Temporal annotations

Example 1

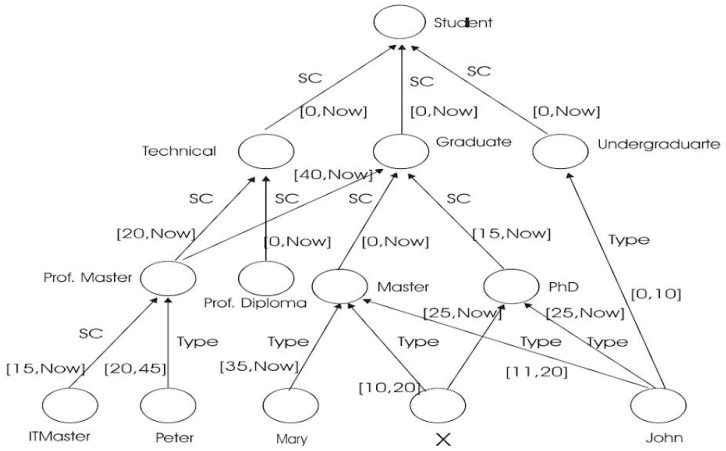
Example from [ANNOTATED RDF]:



Example aRDF graph annotated with $\mathcal{A}_{time-int}$. Extracted from the GovTrack dataset available at http://www.govtrack.us.

Example 2

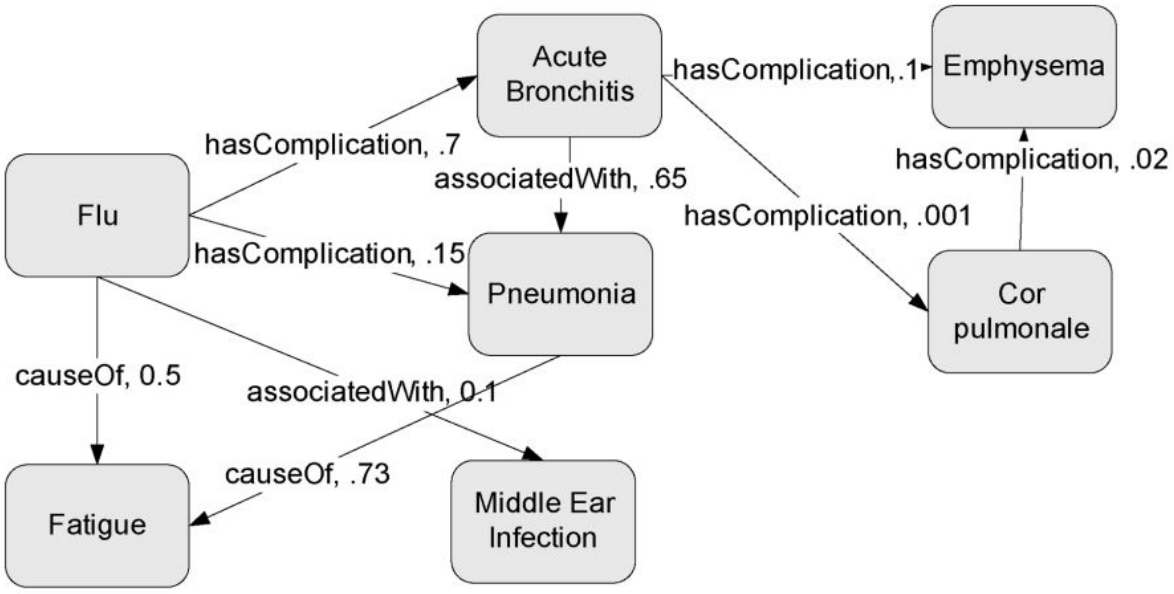
Example from [TEMPORAL RDF]:



A temporal RDF graph accounting for the evolution of the university ontology

Fuzzy annotations

Example from [ANNOTATED RDF]:

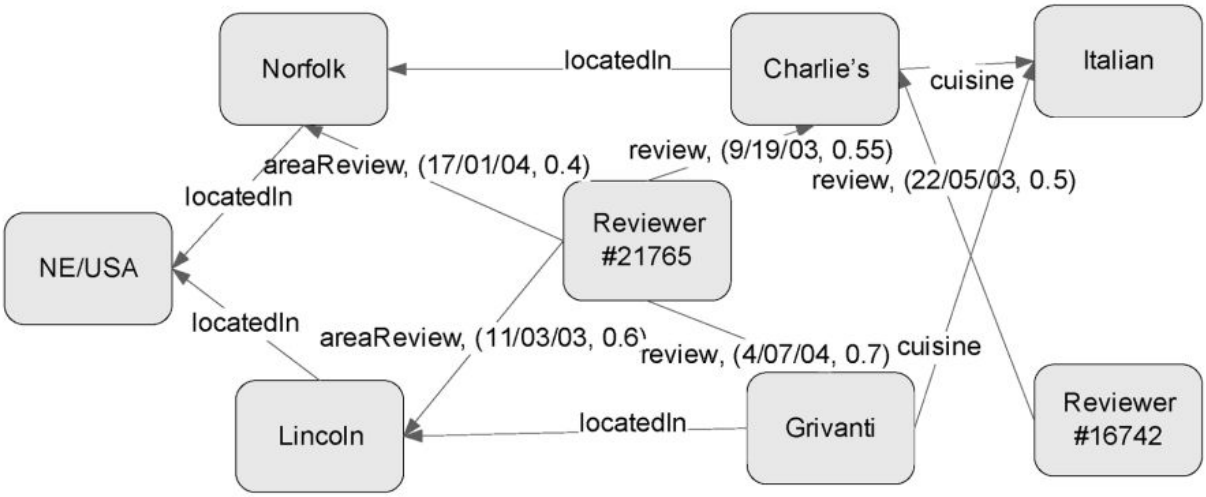


(hasComplication, rdfs:subPropertyOf, associatedWith)

Example aRDF graph annotated with $\mathcal{A}_{fuzzy}$. aRDF constructed based on information from www.wrongdiagnosis.com.

Temporal and Fuzzy annotations (Multiple types of annotations)

Example from [ANNOTATED RDF]:



Example aRDF graph annotated with $\mathcal{A}_{fuztime}$. Extracted from the ChefMoz dataset available at http://chefmoz.org.

GS Cardinality and Keys Report

4a) GS-Keys Overview

Angela Bonifati, Andrea Calì, Stefania Dumbrava, George Fletcher, Alastair Green, Keith Hare, Jan Hidders, Borislav Iordanov, Victor Lee, Bei Li, Wim Martens, Filip Murlak, Josh Perryman, Sławek Staworko, Gabor Szarnyas, Dominik Tomaszuk

Executive Summary

A proposal for property graph key constraints is put forward, based on industry use-cases, the scientific literature, and in-depth debate within a subteam of the LDBC PGSWG consisting of 15 senior industry and academic experts. We propose a key constraint model based on the classic notion of equality-generating dependencies, which generalizes the standard notion of key constraint in earlier data models. Key constraints can be defined on all first-class citizens of a property graph: node objects, relationship objects, and path objects.

In this model, a key constraint *KC* consists of a graph pattern *P*, a set *S* of “selector” elements in *P* and a “target” element *t* in *P*. *KC* holds on a property graph *G* if and only if, for any two matches m_1 and m_2 of *P* in *G*, if $m_1(s) = m_2(s)$ for every element *s* of *S*, then it must be the case that $m_1(t) = m_2(t)$.

For example, suppose *P* is a pattern which just selects nodes *x* labeled `Person`, *S*= {*x*.`employeeID`}, and *t* = *x*. This is a key constraint on node objects which states that `employeeID` property values uniquely identify Person nodes, i.e., for any two `Person` nodes, if they have the same `employeeID`, then they must be the same node.

The model is generic in the sense that it is independent of the language for specifying graph patterns *P* and the semantics of matching graph patterns in property graphs. The choice of pattern language and pattern matching semantics are design decisions, balancing expressivity and computational complexity.

We recommend conjunctive regular path queries (CRPQ) as pattern language and homomorphic matching semantics. CRPQ is the language of subgraph pattern matching queries where edges of patterns are path queries defined by regular expressions over edge labels. An example in a social network property graph: retrieve all people *x*, *y*, and *z* such that there is a path from *x* to *y* using only `Knows` and/or `Likes` edges, and a path from *y* to *z* using only `Follows` edges.

This combination of language and semantics permits highly expressive key constraints while maintaining good computational behaviour. For example, the model can describe constraints that arise from translating relational key constraints (using reasonable relational-to-graph

translations), yet the validation problem (i.e., determining whether or not a given property graph satisfies a given key constraint) is in polynomial time for a large subclass of CRPQ which covers over 99% of graph patterns observed in practice.

Introduction

This document presents a working consensus proposal on *key constraints for property graphs* for the LDBC PGSWG. This document is the outcome of ten conference calls augmented with offline discussions on Basecamp, in the period January-May 2020. The proposal is grounded in a review of the scientific literature and a study of graph key mechanisms used in contemporary systems and frameworks, and reflects in-depth discussion and debate within a subteam of the LDBC PGSWG consisting of 15 senior industry and academic experts.

We motivate and give an informal presentation of the proposal, highlighting main results and contributions, observations, and points for further investigation. This document is a companion to "4b) GS-Keys Formal", which provides a theoretical foundation to the proposal and eliminates any ambiguity that may arise in the informal presentation.

Use cases / requirements

In the following, "attributes" refers to the properties and labels which can be associated with property graph objects, i.e., the properties and labels of nodes, edges, and paths.

We make the following design decisions/requirements. This scoping supports all key constraint mechanisms/use-cases appearing in current practical property graph database systems.

1. The formalism should support defining key constraints for node objects, edge objects, and path objects. Furthermore, the formalism should be extensible and not critically rely on the initial scoping to node, edge, and path objects.

   Rationale. We work in the property graph data model, where nodes, edges, and paths are first-class citizens. We leave the topic of more complex objects (e.g., arbitrary subgraphs) open for later discussion.

2. We view key constraints as a variety of equality-generating dependencies (EGD). In its classical formulation, an EGD is a dependency of the form

   $$\forall\ X\ (\text{if } \varphi(X) \text{ then } x_1 = x_2)$$

   where (a) $\varphi(X)$ is a conjunction of atomic formulas, all with variables among the variables in X, (b) every variable in X appears in $\varphi(X)$, and (c) x_1 and x_2 are distinct variables in X.[1]

[1] For more details see https://doi.org/10.1007/978-0-387-39940-9_1273 and https://en.wikipedia.org/wiki/Equality-generating_dependency

An atomic formula is of the form $P(x_1, \ldots, x_k)$, where P is a k-ary relational symbol, and $x_1, \ldots, x_k$ are variables, not necessarily distinct.

As an example, the following EGD implies that *name* is a "superkey" for the binary relation *employee*(*name, dob*):

$$\forall\ x, y, z\ (\text{if } (employee(x, y)\ \&\ employee(x, z)) \text{ then } y = z).$$

Recall that, as a special case of EGDs, a superkey of a relation r on relation schema $R = \{A_1, \ldots, A_n\}$ is a subset $K \subseteq R$ such that for any distinct tuples t_1 and t_2 in r there is an A in K such that $t_1[A] \neq t2[A]$.[2]

Rationale. Primary keys, candidate keys, and superkeys in the entity-relationship model and relational database model are EGDs.[3] We follow standard intuition of developers and modelers about the notion of "key", accumulated over 4+ decades of practitioner experience with these classic models.

3. We have two types of basic ingredients for defining key constraints:

    a. **Attribute-values** of objects can be compared, both implicitly (e.g., we can check if *x.price = y.price* or we can check if *x.price < y.price*) and explicitly (e.g., we can check if *x.price = 27* or we can check if the label of an edge *x* is "worksAt").

    b. **Object identities** can be implicitly compared for equality (e.g., for variables *x* and *y* bound to objects, check whether or not *x = y*). Object identifiers can not be explicitly observed/compared (e.g., we can not check if *x = 27*, where *x* is an object variable bound, say, to a node). Furthermore, we do not require that object identities and object identifiers can be compared using non-equality predicates (e.g., we do not require that checking *x < y* is possible for object variables *x* and *y*).

    Rationale. It is clear that we need to support observation/comparison of attribute values (i.e., object label values, object property values). Object identifiers (i.e., the physical identity values of objects) are system/implementation specific, whereas the logical notion of object identity is central in defining key constraints for property graphs. Implicit comparison of object identities for equality is a weak form of reasoning over object identity which is strong enough for our purposes. We can not conceive of a practical use case where a key constraint formalism should support reasoning about the internal system-dependent identifiers of objects.

---

[2] This is the definition of "key" as given in Section 1.3 of David Maier, *The Theory of Relational Databases*, Computer Science Press, 1983. http://web.cecs.pdx.edu/~maier/TheoryBook/TRD.html

[3] Note that in this proposal we do not discuss any notion of "primary" key constraints, in the sense of primary keys for the relational model. Similarly, we do not make a distinction between "constraints" and "assertions", nor do we consider practical policies for enforcing key constraints. We just focus here on mechanisms for specifying key constraints (and not for determining which keys are primary and when/how keys should be enforced).

4. We adopt homomorphic matching semantics (instead of isomorphic matching semantics) for the graph patterns used in defining key constraints.

   Rationale. While isomorphism based semantics give a little bit of additional expressiveness, this comes at a rather serious cost of computational complexity (i.e., checking if a key constraint holds becomes harder). We have not yet seen a use case that would justify this additional cost.

   Possible computational complexity issues are: (1) unordered subtree inclusion under isomorphic semantics becomes NP-complete; and we usually assume such problems (problems where we can exploit tree-structure) to be tractable and (2) even finding injective matches of paths is a difficult problem - it's NP-complete in general and even for short paths the technical machinery to do it "efficiently in theory" is very heavy (e.g. color coding).

5. We aim for a simple key constraint formalism which (1) is expressive enough to capture and generalize key constraint mechanisms currently used in practice, yet (2) applies Occam's razor.

   Rationale. We balance a visionary design for the future of property graph key constraints, with a design with good potential for practical impact.

Overview of proposal

We propose a key constraint model based on the classic notion of equality-generating dependencies, which generalizes the standard notion of key constraint in earlier data models. Key constraints can be defined on all first-class citizens of a property graph: node objects, relationship objects, and path objects. Intuitively, a key constraint is (a) a graph pattern, together with (b) a target element of the pattern and (c) a set of "selector" elements of the pattern which constrain the identity of the target.

The model is generic in the sense that it is independent of the language for specifying graph patterns and the semantics of matching graph patterns in property graphs. The choice of pattern language and pattern matching semantics are design decisions, balancing expressivity and computational complexity.

As a core, we recommend conjunctive regular path queries (CRPQ) as pattern language and homomorphic matching semantics.[4] CRPQ is the language of subgraph pattern matching queries where edges of patterns are path queries defined by regular expressions over edge labels. An example in a social network property graph: retrieve all people *x*, *y*, and *z* such that there is a path from *x* to *y* using only `Knows` and/or `Likes` edges, and a path from *y* to *z* using only `Follows` edges.

[4] See Section 3.1.3 of *Querying Graphs*, Bonifati et al., Morgan & Claypool, 2018, https://perso.liris.cnrs.fr/angela.bonifati/pubs/book-Bonifati-et-al-18.pdf

This combination of language and semantics permits highly expressive key constraints while maintaining good computational behaviour. In particular, the validation problem for key constraints (i.e., given a key constraint and a property graph determining whether or not the property graph satisfies the key constraint) is in polynomial time for a large class of CRPQs covering over 99% of graph patterns observed in practice.[5]

Detailed description of the proposal in English accompanied with examples

A key constraint *KC* consists of a graph pattern *P*, a set *S* of elements in *P* (i.e., a set of node variables, edge variables, path variables, and properties of nodes, edges and paths -- used as "selectors"), and a target element *t* in *P*. *KC* holds on a property graph *G* if and only if, for any two matches m_1 and m_2 of *P* in *G*, if $m_1(s) = m_2(s)$ for every element *s* of *S*, then it must be the case that $m_1(t) = m_2(t)$.

For example, suppose *P* is a pattern which just selects nodes *x* labeled `Person`, *S*= {*x*.`employeeID`}, and *t* = *x*. This is a key constraint on node objects which states that `employeeID` property values uniquely identify Person nodes, i.e., for any two `Person` nodes, if they have the same `employeeID`, then they must be the same node.

In the rest of this section we (1) further detail our proposal with additional in-depth examples, many of which are given in the scenario of the LDBC Social Network Benchmark[6]; (2) summarize initial results on the computational complexity of validating key constraints; and, (3) discuss the relationships between key constraints in the relational model and our model for property graph key constraints.

A formal specification of the proposal and further technical discussion can be found in the companion document "4b) GS-Keys Formal".

Key constraint for nodes

1. [using properties only] The "name" property should be a key of countries. More precisely, the name property of a country node should identify the country node. Here, we have the graph pattern *P* = {Country(*x*)}, selector set *S* ={*x*.name}, and the target *t*=*x*.

   We will abbreviate this as

   $$\text{Country}(x)\textbf{:}\ x.\text{name} \rightarrow x.$$

   In pseudocode, we have

[5] See Angela Bonifati, Wim Martens, Thomas Timm: *Navigating the Maze of Wikidata Query Logs* (WWW 2019) and Angela Bonifati, Wim Martens, Thomas Timm: *An analytical study of large SPARQL query logs*. VLDB J. 29(2): 655-679 (2020).

[6] For the LDBC SNB schema, see Figure 2.1 on page 13 of https://arxiv.org/abs/2001.02299

```
WHERE (x:Country)
REQUIRE x.name IDENTIFIES x
```

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 $n_1 = n_2$.

2. [using edges only] Forums can be identified by the posts that they contain, i.e., knowing a post, the forum that contains it is uniquely identified:

   (*x*, *y*, *z*), Forum(*x*), containerOf(*y*), Post(*z*)**:** *z* → *x.*

   In pseudocode, we have

   ```
   WHERE (x:Forum)-[y:containerOf]->(z:Post)
   REQUIRE z IDENTIFIES x
   ```

   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 $n_1 = n_2$.

3. [using properties and edges] Cities are identified by their name and the country they are in. 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:

   (*x*,*y*,*z*), City(*x*), isPartOf(*y*), Country(*z*)**:** *x*.name, *z* → *x.*

   In pseudocode, we have

   ```
   WHERE (x:City)-[y:isPartOf]->(z:Country)
   REQUIRE x.name, z IDENTIFIES x
   ```

   In other words, 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 $n_1 = n_2$.

Key constraints for edges

1. [using from/to nodes] 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:

   (*x*,*y*,*z*), Country(*x*), isPartOf(*y*), Continent(*z*)**:** *x*, *z* → *y.*

   In pseudocode, we have

   ```
   WHERE (x:Country)-[y:isPartOf]->(z:Continent)
   REQUIRE x, z IDENTIFIES y
   ```

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 be the case that $e_1 = e_2$.

2. [using from/to nodes and edge properties] 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, then this edge is identified by the person, the university, and the year:

   (*x*,*y*,*z*), Person(*x*), studyAt(*y*), University(*z*)**:** *x*, *y*.year, *z* → *y.*

   In pseudocode, we have

   ```
   WHERE (x:Person)-[y:studyAt]->(z:University)
   REQUIRE x, y.year, z IDENTIFIES y
   ```

   In other words, given any two edges e_1 and e_2 labeled `studyAt`, 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 `year`, then it must be the case that $e_1 = e_2$.

Key constraints for paths

Suppose that companies directly and indirectly transfer money between each other (initiated by a transfer activity), and it is crucial (e.g., for auditing) that chains of transfer between companies are unique. For a given transfer, the `transferID` property value of each edge in the transfer chain must be the same as the `ID` property value of the transfer activity. We express these requirements as a key constraint on paths:

(*a*, *x*, *c1*), (*c1*, *v* (Transfer {transferID = *a*.ID})*, *c2*),
Activity(*a*), initiatedBy(*x*), Company(*c1*), Company(*c2*)**:** *c1*, *c2* → *v*.

In pseudocode, we have

```
WHERE (a:Activity)-[x:initiatedBy]->(c1:Company)-/v <([:Transfer {transferID: a.ID} ])*> /->(c2:Company)
REQUIRE c1, c2 IDENTIFIES v
```

Computational complexity of key constraints

Our preliminary work on characterizing the computational complexity of key constraints has focused on the problem of **validation** i.e., checking that a given property graph satisfies a set of key constraints. An important objective was to limit the implementation overhead and explore approaches that rely on the available database infrastructure, and the query engine in particular. Our findings are very encouraging: validation w.r.t. a key constraint is reduced to evaluation of a *double query* that consists of a join of two copies of the pattern used in the key constraint, combined with a filter condition consisting of a single inequality of the target terms.

Example. Recall the constraints that requires Forums are identified by the posts they contain, i.e., every post is contained in at most one forum:

```
WHERE (x:Forum)-[containerOf]->(z:Post)
REQUIRE z IDENTIFIES x
```

The corresponding double query identifies the posts that are contained in more than one forums (i.e., identifies violations of the key constraint):

```
MATCH (x:Forum)-[containerOf]->(z:Post)<-[containerOf]-(x':Forum)
WHERE x != x'
RETURN z, x, x'
```

We have investigated key constraints whose pattern has a tree-like shape (is acyclic) since we believe that such patterns will be most commonly employed in practice. We found that although the corresponding double query has no longer a tree-like shape, its structural complexity will nevertheless remain small and will have a relatively low impact on the cost of its evaluation. The double query can be also beneficial for **error reporting** as it allows to identify fragments of the property graph that violate the key constraint. However, more research is needed to investigate ranking methods for presenting violations in an order of relevance/seriousness and methods for aggregating violations that share the same underlying error.

Expressive Power: Graph key constraints versus Relational key constraints

We describe how key constraints in graphs relate to those in relational databases. In order to do so, we need to assume a translation from relational to graph data. So, assume that we translate from the relational model to graphs using a standard translation which is based on pseudo-ER diagrams and which we sketch here:

1. **RELATIONS**. For every relation *R* we introduce a node type *N_R* with properties that contain only those attributes that are not part of any foreign key.

2. **FOREIGN KEYS**. For every foreign key *FK* we introduce an edge type *E_FK* between corresponding node types.

3. **KEY ATTRIBUTES** (or uniqueness constraints, **UC**s). If an attribute of a key (or UC) of a relation *R* is part of a foreign key *FK*, then the corresponding attributes need not be present in the node type *N_R* but will be present in a node type reachable by a path starting with the edge type *E_FK*.

In this case, key constraints from the relational model can be captured in the property graph model, using patterns that are conjunctions of paths (conjunctions of patterns of the form (..)-[..]->(..)-[..]->(..)) that all start with the same node variable and otherwise share no variables. If one uses another standard translation (the Chen-ER based mapping), then one also needs a variation of these patterns that have an edge at the centre instead of a node. In both cases the

patterns are tree-shaped, so the resulting graph key constraints can be validated in polynomial time.

If the relational constraints involve both primary keys and foreign keys, then it may be necessary to have REQUIRE EXACTLY ONE conditions on the graph side. (These constraints are currently not in our proposal, but will be considered when we work on cardinality constraints.) If only primary keys are present, then these conditions are not required.

For further details on the relationships between graph key constraints and relational key constraints, see the companion document “4b) GS Keys, Formal”.

Summary

Main results and contributions

We have introduced a model for specifying key constraints on property graphs. The model balances expressivity and computational complexity, covering contemporary use-cases in practice as well as aiming to support future applications of property graph systems. We establish that our proposal is practical, in both its ability to express sophisticated key constraints on nodes, edges, and paths (including constraints arising from mapping relational data with uniqueness constraints and foreign key constraints into the property graph model), as well as in its well-behaved computational properties.

Observations

We have also considered alternative semantics of matchings of patterns used in key constraints. We have found the standard **homomorphism** semantics to be most natural and enjoying attractive computational properties. **Isomorphism-based** semantics comes with a very high computational cost and a semantics based on **graph simulations**, while having very low computational complexity may be unintuitive and lead to modeling errors.

Open Questions

1. In practice, graph patterns are typically of low complexity, e.g., tree-shaped or with limited cyclic structure. As we have already observed in our preliminary study of the validation problem, such structural complexity has important implications for computational costs associated with key constraints. Further study here is needed to make finer-grained recommendations balancing expressivity and computational complexity. Similarly, study of richer graph pattern languages beyond CRPQ should be undertaken.

2. In our model, we could extend the target from a single target to a complex target, towards more expressive constraints.[7] We leave this open for future discussion. Similarly, further study is needed on extensions/restrictions of the selector set.

3. A number of relevant computational problems remains to be investigated. As with most databases, a graph database is likely to be undergoing frequent modifications, each modification affecting only a fraction of the database contents. Consequently, we plan to investigate the complexity of **incremental validation** beginning with a review of relevant works on materialized view maintenance and how its techniques can be applied for incremental evaluation of the double query. Schemas for property graphs may be large and a result of collaborative effort that may be prone to modeling errors and redundancies. Consequently, we intend to investigate a number of static analysis problems such as checking **consistency** and checking **implication**. However, such problems will be best studied in the presence of schema.

4. We have introduced a pseudocode notation for key constraints, solely for non-normative illustrative purposes. To put key constraints into practice, textual and graphical syntaxes for key constraints should be developed.

5. Other fundamental constraints which arise in practical data modeling include cardinality and participation constraints. The GS-Keys/Constraints team is now turning their attention to such constraints for property graphs.

6. Note that we did not introduce the possibility of NULL property values in our model. The impact of NULLs is a topic for future study.

Conclusion

Key constraints are fundamental in data management. The design and deployment of key constraints for property graphs poses new interesting timely challenges for applications, system engineering, and formal study. The core model presented in this document is put forward as a foundation for accelerating progress along all of these fronts.

4b) GS-Keys Formal

Angela Bonifati, Andrea Calì, Stefania Dumbrava, George Fletcher, Alastair Green, Keith Hare, Jan Hidders, Borislav Iordanov, Victor Lee, Bei Li, Josh Perryman, Wim Martens, Filip Murlak, Sławek Staworko, Gabor Szarnyas, Dominik Tomaszuk

[7] See, for example, Matthias Niewerth and Thomas Schwentick. “Reasoning about XML Constraints based on XML-to-relational mappings”, ICDT 2014: 72-83.

Introduction

This document is a companion to “4a) GS-Keys Overview” and provides a formal description of the proposal for property graph key constraints. The goal is to provide a theoretical foundation to the proposal and eliminate any ambiguity that may arise in the overview document.

Syntax

Let V be a set of variables, A a set of attribute names, T a set of type names, and C a set of constants.

A selector is an expression of the form x or of the form $x.a$, where x is a variable in V and a is an attribute in A.

A graph pattern is a non-empty set of atomic expressions. An atomic expression is an expression of one of the following forms:

- $tp(x)$,
- (x, y, z),
- $(x, (y, r), z)$,
- $x.a \sim y.b$,
- $x.a \sim c$,

where x, y, and z are variables in V; tp is a type name in T; a and b are attribute names in A; $\sim$ is a binary relational operator in the set $\{<, \leq, =, \neq, >, \geq\}$; r is a regular path query (RPQ)[8]; and c is a constant in C. We call the first form a “type pattern”, the second form an “edge pattern”, the third form a “path pattern”, and refer to the remaining forms as “attribute patterns”. Intuitively,

- $tp(x)$ binds to an object of type tp;
- (x, y, z) binds to an edge y from node x to node z;
- $(x, (y, r), z)$ binds to a path y (conforming to r) from node x to node z;
- $x.a \sim y.b$ denotes that the value v_a of attribute a and the value v_b of attribute b, of the objects bound to x and y, resp., satisfy the (in)equality $v_a \sim v_b$;

[8] See Section 3.1.1 of *Querying Graphs*, Bonifati et al., Morgan & Claypool, 2018, https://perso.liris.cnrs.fr/angela.bonifati/pubs/book-Bonifati-et-al-18.pdf

- $x.a \sim c$ denotes that the value v_a of attribute a of the object bound to x satisfies the (in)equality $v_a \sim c$ (in particular, following the definition of the data model, $x.a = lbl$ denotes that the object bound to x has label a).

Essentially, a graph pattern defines a conjunctive regular path query (CRPQ)[9], i.e., a subgraph-matching pattern whose edges are labeled with regular expressions, with additional filters on objects' attributes.

A key constraint is an expression of the form $P : \bar{s} \rightarrow t$, where:

- P is a graph pattern,
- $\bar{s} = (s_1, \ldots, s_n)$ is a tuple of selectors (without repetitions), $n \geq 0$,
- t is a distinguished selector called the target of the constraint,

such that every variable appearing in t or in $\bar{s}$ also appears in P. Intuitively, a key constraint states that "target t is uniquely identified by the identities of objects and the values of attributes selected by $\bar{s}$ in the graph pattern P."

Open design issue A. We do not place any constraints on the structure of the graph pattern P. It is natural to consider restrictions such as the following.

- P contains at most k atomic expressions of the form (x, y, z), for some fixed $k \geq 0$, and no atomic expressions of the form $(x, (y, r), z)$.
    - In some contemporary graph DB systems, $k \geq 0$ when the target is a node and $k = 1$ when the target is an edge.
- The expressions the form (x, y, z), $(x, (y, r), z)$, and $x.a \sim y.b$ in P must form an acyclic structure (i.e., the hypergraph of P is acyclic).
    - This allows us to control the computational complexity of constraint enforcement.
    - For example, in Fact Based Modeling[10], expressions the form (x, y, z) and $(x, (y, r), z)$ must form a set of chains starting from the same variable.
- The atomic expressions of the form (x, y, z) and $(x, (y, r), z)$ in P must form a single connected component.
    - If they are not, then the key probably expresses a non-intended property.

We could also consider intuitive restrictions on $\bar{s}$, such as:

[9] See Section 3.1.3 of *Querying Graphs*, Bonifati et al., Morgan & Claypool, 2018.
[10] http://www.factbasedmodeling.org/home.aspx

- selectors of the form y listed in $\bar{s}$, cannot occur in P in an atomic expression of form (x,y,z), i.e., key constraints defined in terms of edge identity are prohibited.
    - Adopted, e.g., by Fan et al., "Keys for Graphs." PVLDB 8(12): 1590-1601, 2015.[11]

Open design issue B. The atomic expressions over which graph patterns are constructed could of course be extended with additional forms (or, we could consider different query languages altogether).

Semantics

Let G be a property graph and K a key constraint $P : \bar{s} \rightarrow t$. We next define when "$K$ holds on G", i.e., "G satisfies K".

A homomorphism from P to G is a function m from the set of all variables appearing in P to the set of all objects of G (i.e, the set of nodes and edges of G) such that, for each p in P

- if p is of the form $tp(x)$, then $m(x)$ has type tp in G.
- if p is of the form (x,y,z), then $(m(x),m(y),m(z))$ is an edge in G.
- if p is of the form $(x,(y,r),z)$, then $m(y)$ is a path conforming to r from $m(x)$ to $m(z)$ in G.
- if p is of the form $x.a \sim y.b$, then $m(x).a \sim m(y).b$ holds in G.
- if p is of the form $x.a \sim c$, then $m(x).a \sim c$ holds in G.

A K-match of object o in G is a homomorphism m from P to G such that $m(x) = o$, where x is the variable appearing in the target t of K.

K-matches m_1 and m_2 of object o_1 and o_2, resp., coincide if and only if

1. $m_1(x) = m_2(x)$ for each selector of the form x appearing in $\bar{s}$, and
2. $m_1(x).a = m_2(x).a$ for each selector of the form $x.a$ appearing in $\bar{s}$.

K holds on G if and only if for all objects o_1 and o_2 in G, if there exist coincident K-matches of o_1 and o_2 in G, then $o_1 = o_2$ in the case when $t = x$ and $o_1.a = o_2.a$ in the case where $t = x.a$.

Open design issue C. An alternative semantics, adopted by Fan *et al.*[12], is obtained by requiring that K-matches are isomorphisms; that is, that they map different variables to different objects.

[11] http://www.vldb.org/pvldb/vol8/p1590-fan.pdf
[12] Fan et al., "Keys for Graphs." PVLDB 8(12): 1590-1601, 2015.

The isomorphism semantics offers a bit more expressive power: for instance, it allows expressing that each node is uniquely identified by any two of its neighbours. Under the homomorphism semantics this can be expressed only by relying on an already imposed uniqueness constraint (e.g., by referring to an attribute whose value uniquely identifies nodes), but is not expressible in isolation. The (slightly) increased expressive power of the isomorphism semantics comes at a rather serious computational cost (see the next section). Instead of the homomorphism semantics, we could adopt an even weaker tractable notion. For example, we could consider *simulations* of P in G.[13] Simulations can be computed in time cubic in the size of the P and G, but they do not give a very intuitive notion of graph matching. Overall, this open design issue is closely related to the more general question facing the whole PGSWG, namely, "what does it mean for a graph to conform to a schema?". Hence, we leave this open for future discussion.

Open design issue D. We could extend the target from a single target to a complex target, towards more expressive constraints.[14] We leave this open for future discussion.

Complexity

Let K be a key constraint $P : \bar{s} \rightarrow t$. Throughout this section we assume that the selectors $\bar{s}$ and the target t only mention node and edge variables (no path variables). Let $\bar{x} = (x_1, \ldots, x_k)$ be the tuple of all variables occurring in P (without repetitions). For any tuple $\bar{y} = (y_1, \ldots, y_k)$ we write $P(\bar{y})$ for the graph pattern obtained from P by replacing each x_i with y_i. Note that $P(\bar{x})$ is P itself. Similarly, $\bar{s}(\bar{y})$ and $t(\bar{y})$ are obtained by replacing each x_i with y_i in $\bar{s}$ and t, respectively.

We can cast K as the tuple generating dependency $P(\bar{x}) \wedge P(\bar{y}) \wedge \bar{s}(\bar{x}) = \bar{s}(\bar{y}) \rightarrow t(\bar{x}) = t(\bar{y})$ where $\bar{y}$ is a tuple of fresh variables and $\bar{s}(\bar{x}) = \bar{s}(\bar{y})$ stands for the conjunction of equalities $s_i(\bar{x}) = s_i(\bar{y})$ for all $i = 1, 2, \ldots, n$. Consider the query obtained from $P(\bar{x}) \wedge P(\bar{y}) \wedge \bar{s}(\bar{x}) = \bar{s}(\bar{y})$ by projecting out all variables apart from the ones occurring in $t(\bar{x})$ and $t(\bar{y})$. We shall call the resulting binary query the double query of P. Validating K on property graph G amounts to evaluating the double query on G and checking that for each pair (o_1, o_2) of returned objects, $o_1 = o_2$ in the case when the target t is of the form z, or that $o_1.a = o_2.a$ in the case when t is of the form $z.a$. Equivalently, we need to check that the query obtained from the double query by including an atomic expression $\neg\, t(\bar{x}) = t(\bar{y})$ is not satisfied in G; that is, the returned answer is empty. Thus, we have proved the following.

Proposition 1. Key constraints validation reduces to non-satisfaction of graph patterns with a single negation.

---

[13] See Section 7.3.3 of *Data on the web*, Serge Abiteboul et al., Morgan Kaufmann, 1999, https://pdfs.semanticscholar.org/48f9/85d15f797940473600aefedd98c9180d81b7.pdf.

[14] See, e.g., M. Niewerth, T. Schwentick, "Reasoning about XML constraints based on XML-to-relational mappings", ICDT 2014: 72-83.

We can also show the converse (in a sense).

Proposition 2. Let C be a class of graph patterns that is closed under taking unary joins with atomic queries (that is, allow adding atomic expressions sharing a single variable with the graph pattern). Non-satisfaction of graph patterns from the class C reduces to validation of key constraints with graph patterns from class C.

Proof. Consider a query Q and let z be a node variable in Q. Introduce a fresh label L and modify the query so that each node mentioned in the query is required to have this label. Introduce another fresh label L' and extend the query Q so that a fresh node variable z' with label L' is connected to the variable z. Take the key constraint K with the resulting query as the graph pattern, all variables of Q as selectors and the fresh variable z' as the target. Consider a labelled property graph G. Define the modified graph G' obtained by adding two fresh neighbours with label L' to each node. We have that G satisfies the query Q if and only if G' violates the key constraint K.

Given that query evaluation is in general intractable (in terms of combined complexity), Proposition 1 does not give sufficient information about the complexity of validation. Indeed, because in the course of the reduction we replace the graph pattern with the double query, the structural properties of the query are changed. We express structural properties of a graph pattern P in terms of the hypergraph H_P of P whose nodes are the variables used in P and hyperedges are induced by the atomic expressions in P. Even if the original graph pattern is acyclic (in the sense that its hypergraph is Berge-acyclic), its double query need not be so.

Example 1. Consider the key constraint $(x, e, y), (y, f, z) : x, z \rightarrow y$ using an acyclic graph pattern. The corresponding double query is equivalent to $(x, e, y), (y, f, z), (x, e', y'), (y', f', z)$ which is a cycle of length four.

This example shows that the structure can get more complicated when passing to the double query. The good news is that we can quantify this change using the notion of tree-width, and the change is not dramatic. The tree-width of a graph pattern Q, written $tw(Q)$, is the tree-width of its hypergraph H_Q; that is, the tree-width of the Gaifman graph of H_Q, whose nodes are the nodes of H_Q and edges connect nodes co-occurring in a hyperedge of H_Q.

Proposition 3. Let Q' be the double query of a query Q with respect to selectors r and target t. Then, $tw(Q') \leq 2 \cdot tw(Q) + 1$.

Proof. Consider a tree decomposition T for the query Q with bags of size at most $d = tw(Q) + 1$. (Without loss of generality we can assume that the variable used in t is present in the root bag of T.) We can construct a tree decomposition T' for the double query Q' by putting together two copies of T corresponding to the two copies of Q that constitute Q'. More precisely, T' has the same structure as T, and in each node we take the union of the bag from

each copy of T. In the double query, equalities (and attribute equalities) are introduced only between an original variable x_i and its copy y_i. Such pairs of variables will end up in the same bags, so the equality expression will be captured by each of these bags, and T' will indeed be a tree decomposition for the double query Q'. Bags in T' will have size $2d$, which means that Q' has tree width at most $2d - 1 = 2 \cdot tw(Q) + 1$.

While Example 1 shows that the double query of an acyclic graph pattern need not be acyclic, Proposition 3 guarantees that it will have tree-width at most 5, because acyclic graph patterns have tree-width at most 2.

Remark. A tighter measure of the structural complexity of queries is hypertree-width[15], but for graph patterns the gain is small. The reason is that $tw(H) + 1 \geq htw(H) \geq \frac{1}{d}(tw(H) + 1)$ for each hypergraph H with only at most d-ary hyperedges, and the hypergraphs of every graph pattern has only at most 3-ary hyperedges.

Graph patterns of bounded tree-width can be evaluated in polynomial time. Combining Propositions 1 and 3 we get the same result for validation of key constraints.

Corollary 1. Key constraints using graph patterns of bounded tree-width can be validated in polynomial time.

In the special case of acyclic graph patterns, the algorithm is easy to visualise. The variables of an acyclic graph pattern can be naturally arranged into a tree such that each binary atomic expression involves neighbouring variables, and each ternary atomic expression involves variables that form a path of length 2; we can assume that the variable used in the target is the root of this tree. We can evaluate the double query efficiently using a dynamic algorithm that processes the query bottom up, computing pairs of objects selected by the subquery of the double query induced by the subtrees rooted at the variables x_i and y_i. The complexity of the algorithm is quadratic in the number of objects in the graph (up to logarithmic factors), which is optimal given that we are evaluating a binary query. This algorithm can be seen as a special case of the general dynamic algorithm for queries of bounded tree-width. The algorithm works for CRPQs as well, because the constituting RPQs can be precomputed and treated as atomic edges (not that this affects the number of objects in the graph).

The computational properties of key constraints under the isomorphism semantics are much worse. Even finding injective matches of a path in a graph is NP-complete, as it generalizes the Hamiltonian Path Problem. Since Proposition 2 holds also for the isomorphism semantics, validation of key constraints whose graph patterns have the structure of a path would be intractable under the isomorphism semantics. Finding injective mappings of paths is fixed-parameter tractable (with the length of the path as the parameter)[16], but the algorithms rely

[15] See G. Gottlob, N. Leone, and F. Scarcello, “Hypertree decompositions and tractable queries”, Journal of Computer and System Sciences, 209:1–45, 2002.
[16] J. Plehn, B. Voigt, “Finding minimally weighted subgraphs”, in Graph-Theoretic Concepts in Computer Science, pages 18–29, 1990.

on heavy machinery (e.g. color coding)[17]. On the other hand, allowing inequalities in graph patterns does not invalidate the approach sketched above, as long as inequalities between variables are treated just like other atomic expressions (that is, potentially increasing the tree-width of the query). Note, however, that while the existence of an injective match of a path of length k can be expressed with inequalities as a query of tree-width k, the polynomial algorithm for queries of bounded tree-width does not show that the existence of an injective match of a path of length k is fixed-parameter tractable with parameter k, because it runs in time $\tilde{O}(n^k)$ on the graph of size n.

The approach to validation described above can be used to support rudimentary error reporting. Indeed, it is natural to see each pair of different nodes selected by the double query as a violation of the key constraint. The system could report these pairs to the user, possibly justifying each pair with an arbitrarily selected match of the double query. In some scenarios, the number of such matches might be interpreted as the strength of the evidence that the two elements should be merged, in the spirit of entity resolution.

Expressiveness: Graph Key Constraints vs Relational Key Constraints

We illustrate the constraints that are needed to express relational constraints by means of an example. Consider the following relational schema where **UC** represents a uniqueness constraint and **FK** a foreign key constraint. To keep the example simple we will assume there are no null values.

- R1(a, b, c, d, e) with **UC**{a, b, c, d} and **FK** fk1= R1[b, c] ⊆ R2[b, c] and **FK** fk2 = R1[c, d] ⊆ R4[c, d]
- R2(b, c, f, g) with **UC**{b, c} and **FK** fk3 = R2[c, f] ⊆ R3[c, f]
- R3(c, f, h) with **UC**{c, f}
- R4(c, d, i) with **UC**{c, d} and **FK** fk4 = R4[d, i] ⊆ R5[d, i]
- R5(d, i, j) with **UC**{d, i}

This translates under the pseudo-ER mapping to a graph schema with the following node and edge types:

```
r1 = :R1 {a DOM, e DOM}

r2 = :R2 {b DOM, g DOM}

r3 = :R3 {c DOM, f DOM, h DOM}
```

[17] N. Alon, R. Yuster, U. Zwick, “Color-coding”, J. ACM, 42(4):844–856, 1995.

```
r4 = :R4 {c DOM}

r5 = :R5 {d DOM, i DOM, j DOM}

fk1 = (r1)-[:FK1]->(r2)

fk2 = (r1)-[:FK2]->(r3)

fk3 = (r2)-[:FK3]->(r3)

fk4 = (r3)-[:FK4]->(r4)
```

Note that columns that are the source of a foreign key, such as {b, c} in R1 are not mapped to properties since they are redundant. In addition the translation requires the following graph constraints:

To represent **UC**{a, b, c, d} for R1:

```
WHERE (r1 :R1)-[:FK1]->(r2 :R2)-[:FK2]->(r3 :R3),
      (r1 :R1)-[:FK3]->(r4 :R4)-[:FK4]->(r5 :R5)
REQUIRE r1.a, r2.b, r3.c, r5.d IDENTIFIES r1
```

To represent **UC**{b, c} for R2:

```
WHERE (r2 :R2)-[:FK2]->(r3 :R3)
REQUIRE r2.b, r3.c IDENTIFIES r2
```

To represent **UC**{c, f} for R3:

```
WHERE (r3 :R3)
REQUIRE r3.c, r3.f IDENTIFIES r3
```

To represent **UC**{c, d} for R4:

```
WHERE (r4 :R4)-[:FK4]->(r5 :R5)
REQUIRE r4.c, r5.d IDENTIFIES r4
```

To represent **UC**{d, i} for R5:

```
WHERE (r5 :R5)
REQUIRE r5.d, r5.i IDENTIFIES r5
```

To represent **FK** fk1:

```
WHERE (r1 :R1)
REQUIRE EXACTLY ONE f SUCH THAT (r1)-[f:FK1]->()
```

To represent **FK** fk2:

```
WHERE (r1 :R1)
REQUIRE EXACTLY ONE f SUCH THAT (r1)-[f:FK2]->()
```

To represent **FK** fk3:

```
WHERE (r2 :R2)
REQUIRE EXACTLY ONE f SUCH THAT (r2)-[f:FK3]->()
```

To represent **FK** fk4:

```
WHERE (r3 :R3)
REQUIRE EXACTLY ONE f SUCH THAT (r3)-[f:FK3]->()
```

To represent that properties that correspond to columns linked by **FK** fk1, **FK** fk2 and **FK** fk3 must have the same value:

```
WHERE (r1 :R1)-[:FK1]->(r2 :R2)-[:FK2]->(r3 :R3),
      (r1 :R1)-[:FK3]->(r4 :R4)
REQUIRE r3.c = r4.c
```

One sees that the patterns that are necessary to capture relational keys are “octopus patterns”. An octopus pattern is a pattern that consists of a conjunction of path patterns (of the form (..)-[..]->(..)-[..]->(..) etc) that all start with the same node or edge variable and otherwise share no variables. An example is the pattern we see in the first constraint in the example:

```
WHERE (r1 :R1)-[:FK1]->(r2 :R2)-[:FK2]->(r3 :R3),
      (r1 :R1)-[:FK3]->(r4 :R4)-[:FK4]->(r5 :R5)
```

Note that octopus patterns are acyclic, which means that the resulting key constraints can be validated efficiently, as discussed in the previous section.

A more formal description of the relational-to-graph transformation that we considered is described next:

1. **RELATIONS**. For every relation R we introduce a node type N_R with properties that contain only those attributes that are not part of any foreign key.
2. **FOREIGN KEYS**. For every foreign key FK we introduce an edge type E_FK between corresponding node types.
3. **ATTRIBUTES KEY ATTRIBUTES** (or **UC**s). If an attribute of a key (or UC) of a relation R is part of a foreign key FK, then the corresponding attributes need not be present in the node type N_R but will be present in a node type reachable by a path starting with the edge type E_FK.
   Things to keep in mind:

   a. FK chains of length greater than one;
   b. FK chains that visit the same relation.
4. **UNDERSPECIFIED**. What happens with a cyclic FK, for instance, consider R(a,b) with UC{a}, P(a,c) with UC{a}, and two FKs R[a] ⊆ P[a] and P[a] ⊆ R[a]
   A SOLUTION: an arbitrary relation is chosen and the key attribute included in the corresponding node type.
5. **REIFICATION OF BINARY RELATIONSHIPS**.
   Relations that are used to represent a non-functional relationship between two entities (in Chen-like diagram) will be translated to a node type and two edges types, while a single edge type would suffice and be potentially more natural. This is minor and one should be able to alter the translation to handle such relations accordingly but we refrain from doing in the interest of keeping things simple

WG3:MMX-076 LDBC PGS:AG-12

LDBC Property Graph Schema Working Group

Proposal for requirements, scope and roadmap

Alastair Green, first draft 25 May, second draft 16 June 2020.

My thanks to Bei Li, Jan Hidders and Keith Hare for their useful corrections and comments on the first draft. The resulting document is a proposal for discussion (including with WG3, our liaison partner), and a guide to some shared thoughts in the LDBC's PGS working group, but does not claim to represent a formed consensus in the group.

This group (PGS WG) began life as a self-organized group of interested researchers and practitioners, supportive of the GQL initiative, who met in Berlin at the close of the W3C workshop on graph data management standards in March 2019.

Headed initially by Juan Sequeda (data.world), and now jointly with Jan Hidders (Birkbeck University of London), the PGS WG was reconstituted as an LDBC working group in July 2019, alongside the Existing Languages Working Group (ELWG). In February 2020 the GQL Formal Semantics Working Group was also chartered by the LDBC board.

All three groups are community efforts created to **support the formation of an effective GQL standard database language for property graph databases**.

Being part of LDBC has enabled information sharing, because of LDBC's Category C liaison with WG3. This allows the LDBC working groups to comment on working documents and discussion papers, and to get early sight of WG3's emerging consensus. It also allows LDBC to make proposals that run ahead of existing WG3 work, and to review the features and design of a draft GQL language through the alternative lens of formal language specification.

The Property Graph Schema Working Group has set out to address requirements (and constraints) for schema, in an order, and with initial self-constraints, to allow incremental progress to be made.

Summary list

R17. Closed and open schema models should be defined as parts of a single schema model.

R18. The relative market failure of SQL 1999 user-defined structured types should not be used to exclude subtyping a priori, but it is an object lesson.

R19. GQL schema designs from the PGS Working Group should not be constrained or determined by the schedule, priorities or scope of a particular edition of GQL.

Ordering of, and self-imposed constraints on, the work of the PGS WG in its first phase (January to August 2020)

OC 0. Defining a closed (wholly prescriptive) and open (non-prescriptive) schema model first.

OC 1. Defining a schema model without subtyping first.

OC 2. Using the Extended Entity Relationship Model as a measure of scope and success

OC3. Producing detailed contributions to WG3 and investing time in explaining and discussing them before expanding scope.

OC4. Not attempting to define mappings from other schema models to the GQL schema model.

OC5. Not attempting to define a pan-graph schema model to sit above the GQL schema model.

Requirements and constraints for the PGS design of GQL schema

R0. GQL schema is determined and constrained by the GQL standard property graph model.

LIke everything else in GQL, the property graph model is not decided until the ink is dry on the final standard. And the PGS WG should propose change or innovation, even at the most basic level of the data model, when it can be justified.

However, the role of a schema model and any associated schema languages is to describe and prescribe metamodels (application domain data models) which specialize the property graph data model. We need a stable definition of the data model to build upon.

The property graph data model is an industrial fact, and an object of research, but is not yet standardized: there are many interpretations. That plurality does not mean that we are starting from nowhere, and one purpose of our work is to help end (complete) discussion on the data model, in the GQL universe of discourse.

The property graph model was in good part defined negatively, as “not RDF”. In particular property graphs are not about semantics, nor are they of the Web. They are abstract data structures, which do not inherently hold or represent Web resources, and which therefore inherit no intrinsic or general notion of resource identity and identification. Providing the means for cataloging or discovering the semantics of their information content is uninteresting, or beyond the core ambitions of property graph technology.

Property graphs

i. limit the functionality of RDF by preventing graphs or graph elements being used as nodes, edges or property values (so in a property graph it is not possible to “make a statement about”, i.e. create an edge between two edges, or a node and an edge, or an edge and a named graph).

ii. extend the functionality of RDF by allowing sets of properties to be associated with both nodes and edges[1].

[1] The RDF* proposal extends RDF to allow properties on edges to be defined, but this is not part of the W3C suite of standards yet. It is true that an RDF object can be a value of a complex data type, and that in that sense a set of properties, or nested sets of properties, can be attached in one go to a node. In property graphs a node or edge is inherently a (possibly empty) set of properties and/or labels, so the containment or association of attributes with a graph element is a special, distinguished relationship.

iii. simplify the graph topology by comparison with RDF by reducing the number of nodes and edges by treating properties as being “off the graph”.

iv. complicate the graph topology by allowing multiple edges between two nodes, and by allowing isolated nodes.

Our starting point for a standard is to agree on some reasonable superset of all “wild” variants,so that GQL’s property graph data model is a conservative progression or extension, not an obstacle or a destructive revolution, for existing products and theories which self-identify as implementing or using property graphs.

The process of supersetting helps the standard to *leave behind* accidental restrictions, implementation quirks, competitive differentiators, and groupthink usage assumptions. It also *preserves* common and distinguishing features.

Languages, products and projects which have adopted the name of “property graph”, including Neo4j and Cypher, Tinkerpop, PGQL, TigerGraph and GSQL, and SQL-PGQ are all subsumed by a data model where a graph can have these features

1. Mixed (directed and undirected)
2. Multigraph (multiple edges between two nodes)
3. Labels and properties are equally applicable to nodes and edges, in any quantity
4. Properties can have a scalar value or a collection of values
5. Properties can also have associated properties

Equally, those features which characterize the property graph model are highlighted, which cannot be eroded without voiding the distinct nature of the model:

1. Graphs, not hypergraphs, with binary, not n-ary, relationships.
2. Nodes and edges can be attributed (have data values associated with them that are not nodes and edges)
3. Attributes are not mandatory (so pure topological graphs can be stored and queried)
4. Attributes can be limited to atomic types and limited in cardinality, so that a user can define graphs that are “pure”, where complex data structures are only

modelled using nodes and edges. While possible, this is an unusual usage pattern, because:

5. **Attributes can also be used to create data structures that are not arbitrary graphs, allowing the coexistence of two data models, one for graphs and one for attributes on graph elements**.

The last feature is what makes a property graph out of a graph.

The **not-arbitrary-graph data structures** may be pretty simple (collections, records), or quite complex, but to create a data model distinct from pure graphs, they must be **more complex than a single atomic value, and less complex than an arbitrary graph**.

The justification for moving some data into limited-graph data structures is that it allows the graph topology to stand out from the details of data items, in the same way that we talk about “not seeing the wood for the trees”. This has proved useful and attractive in the market.

R1. GQL schema should provide a way of describing the topology and values to be found in a property graph.

Property graph database products and OSS projects, and their associated languages, have formed a commercial and engineering reality without any standard approach or even a common rough consensus on schema.

1. The main reason for this is the origin of the property graph database category as an outgrowth of **Java libraries which model a graph data structure**. (See “An overview of the recent history of Graph Query Languages”, WG3:YTZ-029R1, Tobias Lindaaker, 2018.)

2. Starting with Neo4j [Network Engine for Objects for Java], which was forked via the shared Blueprints API into Tinkerpop, libraries with Java APIs a) depended on Java’s type system, and b) were used in programs written in a language with strong typing. The need for a separate type system to describe graphs as complex data structures was not felt strongly.

3. Graph databases were also seen as a pragmatic way of **mashing up data from multiple disparate sources** in web servers hosting applications written in Java. In that environment a premium was placed on ease of integration, speed of development and an Agile approach to incremental feature evolution. (These drivers were reflected in other concurrent technologies such as JSON, and NoSQL movement [including the Hadoop family], which emerged and matured

between 2005 and 2015. The flexible datatype system of SQLite is a parallel manifestation.)

4. A third reason for going “schema free” or “schema lite” was to avoid the **complexity of the RDF schema/ontology ecosystem**, which was perceived as entangled, Web-specific, hard for mere mortals to understand, and leading to over-engineered data models which are too hard to maintain. Property graphs just ran round all of that.

In a second phase, property graph databases began to change.

A. Embedded libraries were replaced by a **client-server model**, severing the tie between the application programming language and the data engine. This eroded the “JVM assumption” and opened the road to native language servers.

B. **Declarative query languages** like Cypher and PGQL began to emerge, followed by SQL/PGQ and GSQL which were again important for increasing “language genetic diversity”.

C. Use of property graph databases has increasingly moved into **the enterprise mainstream**, fuelled by the leading edge use of graphs as a central organizing principle for data in major tech services like search and social networks. Emil Eifrem at Neo4j describes this as the “democratization of graph data”: it’s not just for Google or Facebook. Network analysis in finance, telco, retail, insurance, biomedical, pharmaceutical, intelligence and military systems, revolving around natural and social behavioural pattern detection and resource optimization has fuelled a growth that is increasingly intersecting machine learning-based data science[2].

[2] It is interesting to note how large a divergence there is between this growing reality and the consensus view of many leading figures in the database industrial-research complex, in which graph databases figure as an invisible or barely discernible and irritating rash on the vast posterior of the relational database industry. See “The Seattle Report on Database Research”, 2018, and input papers.

The debated need for a network model, and the relationship between it and the relational model, is a very old discussion (see Codd, E. F. 1971. “Normalized Data Base Structure: A Brief Tutorial.” In *Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control*, 1–17. SIGFIDET ’71. New York, NY, USA: Association for Computing Machinery). Suffice it to say, for our current purposes, that graph databases have found a place in the market aligned with Codd’s observations that:

> “It may be argued that in some applications the problems have an immediate natural formulation in terms of networks. This is true of some applications, such as studies of transportation networks, power-line networks, computer design, and the like. We shall call these network applications and consider their special needs later. The numerous data bases which reflect the daily operations and transactions of commercial and industrial enterprises are, for the most part,

All of these trends point to an increased need to have a way of capturing the shape and potential values of a property graph in a way that is

a. Independent of general-purpose programming language,

b. Relates closely to declarative querying, and

c. Allows different kinds of users to share a common understanding of a data model.

All three of these trends also bring property graph databases closer to the model of an SQL database.

R2. While being graph specific, and without limiting innovation, GQL schema should play the same broad role as an SQL schema.

SQL is a massively successful standard which has generated a viable and broadly understood mental model of how applications interact with and use tens if not hundreds of database engines. GQL was motivated as a companion standard to SQL, which would share, or borrow and adapt, much from SQL, and avoid idle variation from SQL.

A GQL schema facility should allow **types and instances of graphs** to be defined in a special-purpose database called **a catalog**, which is a common repository for all type and constraint (metadata) definitions, and for instance (data) declarations, using a **Data Definition Language** that is distinct from sub-languages for querying and data manipulation.

R3. GQL must define a data type system, and its schema model must harmonize with that type system.

A distinct database language must define the data types of the values that it stores, and is unlikely to be able to do so by reference to a prior standard. Schema straddles types, constraints and instances, and must mesh perfectly with the data type system.

R4. GQL's type system must not contradict the core type system of SQL.

GQL was constitutionally motivated to be a companion to SQL, and to specifically avoid the mistake of W3C standards which varied at the most basic level of predefined (atomic) datatype definitions.

---

concerned with non-network applications. To impose a network structure on such data bases and force all users to view the data in network terms is to burden the majority of these users with unnecessary complexity."

.

R5. GQL schema semantics should be described in formal mathematical terms.

There is an attempt to describe and verify the GQL language design in formal terms, exemplified by the query language semantics work of the Formal Semantics Working Group. PGS WG participants have voiced the equal importance of formalizing schema/typing designs.

R6. GQL schema should be described in informal terms that aid intuitive understanding in the widest audiences.

The experience of all attempts to create a consensus on describing and defining the limits and shape of a complex technical field is that a combination of informal (intuitive) and formal methods is important for creating shared understanding and precision, particularly when participants come from several different primary backgrounds (theoretical and applied computer science, software engineering, conceptual modelling, end-use, standards authoring/editing etc).

R7. GQL schema should be described with motivating use cases and examples.

Similarly, examples revolving around motivating use cases, are important tools of exposition and of verification.

R8. GQL schema must have a concrete syntactic expression in a schema definition sub-language.

Examples need to show proposed semantics using shared concrete syntax. The PGS WG is not primarily focussed on the final syntax of a schema definition sub-language or languages within GQL, but is a useful testbed. It is helpful if the WG's chosen syntax follows the design philosophy and emerging approaches of work in WG3, where possible.

R9. GQL schema must integrate with the GQL catalog, and the schema definition language must be a sub-language of the DDL used to maintain the catalog.

It is assumed that GQL schema definition language(s) will be used to maintain an SQL-style catalog of metadata and data objects.

R10. GQL schema should harmonize (semantically and syntactically) with the emerging GQL query language, and therefore with SQL/PGQ's pattern-matching sub-language.

SQL/PGQ is the most mature aspect of GQL. It contains SQL-specific parts (graph views over tables and the GRAPH_TABLE function) and a graph pattern-matching sub-language based on Cypher and PGQL.

It solidifies the central role of node-edge ASCII line-art pattern representations in the projected mental model and syntactic persona of GQL.

The patterns used for querying data have the same shape as the patterns used for inserting data in Cypher, and that meta-pattern can be extended naturally to schema. The patterns used to insert data describe graph topology and element attribution, with data values. The patterns needed to describe or prescribe the kind of graph topology and element attribution to be found in a graph are almost identical: they differ only in specifying data types for attributes.

It follows that at least the abstract model of GQL schema must cleave to the abstract model of GQL data insertion, and that it is likely and desirable that the concrete syntax for schema, insertion, and query patterns will have a unified design.

R11. GQL schema should provide a way of prescribing the topology and values to be found in a property graph.

A schema in SQL dictates the permitted content of a database. It is *closed-world*: no piece of information not modelled by a specific schema (data model) can be included as a data item. The TigerGraph graph database and its language GSQL share this property with SQL.

There are advantages in being able to rely on an enforced closed schema for data modellers, and application developers, Closed schema also enables compile-time (static) inference of types, allowing higher degrees of query and storage optimization.

The framing documents and the joint statement of Neo4j and TigerGraph proposing the Features and Scope of GQL [reference] state that GQL should be **capable of** imposing the same **closed schema** model as SQL, and should (by implication) therefore not prevent an existing product with that model from adopting GQL. Emulating SQL in this regard has proven market appeal for some users and customers.

R12. GQL schema should not be imposed as a prerequisite for the adoption of the GQL query language by a graph data management product.

Conversely, the ability to operate freely without the constraints imposed by a closed schema is equally valuable. For some databases and/or applications, and for some experimental phases of other databases/applications, the imposition of closed schema is a burdensome overhead or is antithetical to the purpose of the designers. Products such as Neo4j and the OSS Tinkerpop project have operated successfully and indeed attracted market share, precisely because they do not demand up-front determination of a schema which is then enforced at runtime.

GQL should not be defined so as to require that a technically and commercially viable implementation has a closed schema model. Indeed, it should be **possible to operate entirely without a schema**. The bounding requirement on a GQL implementation is that the query engine can operate on data that is presented **in conformance with the GQL property graph data model**, even if the engine is entirely **ignorant of any GQL schema model**.

This could be rephrased with equivalent effect as permitting operation of a GQL query engine in the presence of an **open-world schema model**, that is to say, a model where any data item can be present irrespective of all schema statements.

R13. GQL schema should reflect advances in research and industry since SQL 1999

SQL's schema mode (which has not changed in a major way since SQL 1999) is not the last word in schema theory or practice. Over the last two decades programming language designers have been working to extend their type systems to incorporate a trio of concepts (abstract data type mixins as distinct from behavioural inheritance, a.k.a. data subtyping, integrated with union types and intersection types).

This kind of work can reasonably inform GQL schema design.

Another example is consideration of the need for, and relationship between, nominal and structural typing. SQL is nominal-type-centric. Should GQL follow the same model?

R14. GQL schema should reflect and exploit the property graph data model

This might seem tautologous, or a restatement of R0 and R8, but is a distinct requirement of ambition. The property graph data model is very similar to the Entity Relationship Model/UML class diagrams, often supersets that of RDF, and can be used to approximate the fact-granularity of ORM.There is an opportunity to produce a schema model which exploits the richness and flexibility of the PG data model, and leverages work on schema or conceptual modelling for these other models.

R15. GQL schema should not be limited by the functionality of existing property graph data engines

Existing property graph engines are frequently very limited in their schema capabilities. A property graph schema model that minimally matches the level of schema control expected by SQL database users will be the product of innovative design and will not arise from codification of the superset of industrial practice.

R16. GQL schema should be inclusive of the functionality of existing property graph data engines.

Any elements of schema that have made it into existing engines are valuable precedents.

R17. Closed and open schema models should be defined as parts of a single schema model.

No graph database today allows administrators to pick the proportions of “water” and “ice” they want in the mix of a graph schema. You can run free, or you can be locked down, but you can’t occupy a mid-point on the spectrum between those two ends, in a systematic, well-defined fashion.

The concept of mandating, permitting or blocking subtyping is familiar from OO languages which allow classes to be abstract or concrete (instantiable), and final (closed) or extensible.

We aim for a unified schema model that permits four modes of prescription:

Not prescriptive (open): a user or application can modify a database to contain any object, in any combination or quantity permitted by the property graph data model.

Selectively prescriptive (optional, or open-plus): While retaining the full freedom of the open model, a user can also define the type, combinations and cardinalities of some classes of object which can be instantiated by modification of a database, so that objects can be created which are checked for their conformance to rules in a user-defined schema.

Partially prescriptive (extensible, or partial): Any object described by schema is capable, in principle, of being extended or specialized, and no object or combination of objects that is not at least partially described in a user-defined schema can be instantiated by mutation of a database. This relates to the concepts of abstract and final classes in OO.

Wholly prescriptive (closed): A graph can only be created which wholly conforms to a set of rules (types and constraints) in a user-defined schema. This is the model of SQL.

R18. The relative market failure of SQL 1999 user-defined structured types should not be used to exclude subtyping *a priori*, but it is an object lesson.

User-defined structured types (UDSTs) in SQL 3 (SQL 1999) sought to import Java subclassing into SQL.

Interface mixins or struct mixins point away from behaviour (towards data structures), towards multiple inheritance, and are a feature expected of most modern programming languages (Scala, Rust, Julia, Swift, Go).

The following proposition should be critically examined: “The baby of subtyping should not be thrown out with the bathwater of subclassing.”

SQL UDSTs are interesting because they prototype a model where any expression can be assigned to any target value, so long as the type of the target is a super-type of the type of the expression, *where all super- and sub-types are predefined*. This should not be confused with a partial schema model.

Subtyping is a syntactic compression; it is a conformance accelerator. It is not existential to a valid schema model. SQL-92 and GSQL show this.

R19. GQL schema designs from the PGS Working Group should not be constrained or determined by the schedule, priorities or scope of a particular edition of GQL.

The PGS WG has an opportunity to define a vision and roadmap for GQL schema.

WG3 has the obligation and need to shape a schedule, influenced by priorities arising from implementer/vendor input and the timetable and demands of the standard formation process.

The PGS WG should not attempt to second-guess the prioritization and phasing decisions of WG3, nor should it be limited in its thinking and outputs by those decisions.

Ordering of, and self-imposed constraints on, the work of the PGS WG in its first phase (January to August 2020)

OC 0. Defining a closed (wholly prescriptive) and open (non-prescriptive) schema model first.

Without a clear understanding of what a closed schema model looks like, it is premature to give details of how optional or partial models would work. The design of closed schema should impose no constraints on operating with no schema or an open schema model, and to that extent open schema should also be defined in this first phase.

OC 1. Defining a schema model without subtyping first.

Subtyping introduces complexity. An integrated design for subtyping-free schema including the data model, types system and core constraint system is the first priority.

Subtyping is a powerful abstraction, which can reduce the footprint of a schema declaration in a schema definition language, and is ubiquitous in general-purpose programming languages. It would be wrong to exclude subtyping from a modern schema language.

This is an ordering issue, not a scope issue. Subtyping will be addressed.

OC 2. Using the Extended Entity Relationship Model as a measure of scope and success

EER and UML class models are prevalent in conceptual data modelling. The underlying data model is the same as the core of the property graph data model (with the exception of n-ary relationships).

SQL schema requires a conceptual-logical model mapping that is not needed in most situations using the property graph data model. Being able to map from EERM/UML models to GQL schema would be a big win for GQL.

SQL schema does not provide support for aspects of EERM/UML that are very naturally allied to a property graph schema, like relationship cardinalities.

EERM deals with common subtyping patterns that are a good test of any subtyping facilities proposed for GQL schema.

OC3. Producing detailed contributions to WG3 and investing time in explaining and discussing them before expanding scope.

The primary purpose of the WG is to help make a good GQL standard. The interaction with WG3 will take significant effort, incurring an expository and procedural overhead, and also requiring the evolution of a working method for both groups.

If WG3 finds the work of the PGS WG helpful, then the PGS WG has satisfied a critical success criterion.

Even if WG3 is unable to directly consume or utilize the designs of the PGS WG, it is an independent achievement for the working group to define such designs in a collective effort spanning many participants. Producing the schema equivalent of the G-CORE paper would be a powerful justification, on its own, for the investment of effort by PGS WG participants.

OC4. Not attempting to define mappings from other schema models to the GQL schema model.

OC0, OC1 and OC3 comprise a very large and complex scope. In line with the Berlin workshop consensus, the property graph side of the graph data family needs to define its own standards before addressing standards/model mappings.

OC5. Not attempting to define a pan-graph schema model to sit above the GQL schema model.

The same applies in this regard. A model for arbitrary data types, like mm-ADT or Tinkerpop 4.0 or APG, may be defined, of which GQL schema is at least a partial subset.

The PGS WG does not attempt to replicate such work. There are aspects of schema, such as constraints, that may be more prominent in the GQL project than in those efforts. Mutual awareness, and an appetite for alignment, are valuable aspirations.

Appendix

Slides for presentation and discussion of the preceding papers at WG3

GS-Basic

WG3 report: 24 June 2020

Jan Hidders

Goal of GS-Basic

- General goal
  - Discuss alternatives for the syntax and semantics of basic property-graph schemas / types
- Scope of current discussion:
  - Focus on graph-level (so not attribute level) data model
  - Leave inheritance aside for the moment
  - Leaven open types (at attribute level and graph level) aside for the moment
  - But to some extent subtyping is discussed

Goal of presentation

- Provide an overview of the types of semantics for basic graph types that are on the table in GS-Basic
- Give insight into the ongoing discussion in GS-Basic
- Getting feedback on this discussion

For the full report see:

- **LDBC PGS-B:BAS-01r1** “GS-Basic Overview report”
  - A summary report with emphasis on informal explanation and examples.
- **LDBC PGS-B:BAS-02r1**, “GS-Basic Formal definitions report”
  - A companion report containing fully formal definitions and discussions.

Three proposals and a concern

Proposals:

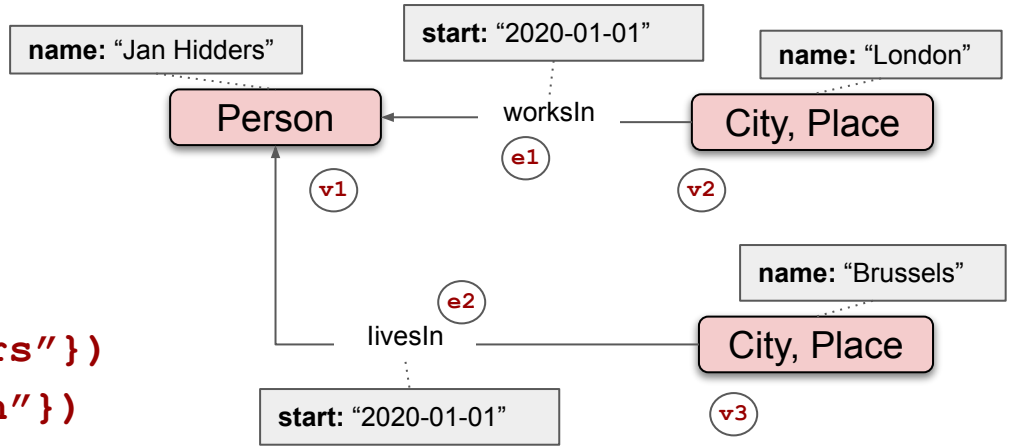
1. The at-least-one-match semantics
2. The exactly-one-match semantics
3. The homomorphism-based semantics

Concern that touches all proposals:

1. Isolation-awareness

Basic Concepts

Property graphs



Our concrete syntax for examples:

```
(v1 :Person {name = “Jan Hidders”})
(v2 :City Place {name = “London”})
(v3 :City Place {name = “Brussels”})

(v1)-[e1 :worksIn {start=“2020-01-01”}]->(v2)
(v1)-[e2 :livesIn {start=“2015-01-01”}]->(v3)
```

Element types

- Vertex types:
  - Concrete syntax: `(:City Place { name STRING, url URL })`
  - Equivalent to: `(: {City LABEL, Place LABEL, name STRING, url URL })`
    - Labels are treated as attributes with a “dummy value” `lbl`
    - The record type here is also referred as a **content type**
- Edge types:
  - ```
    (:Person { name STRING, birthdate DATE })
      -[:livesIn { start DATE}]->
    (:City Place { name STRING, url URL })
    ```
  - Consists of (1) tail vertex type, (2) the content type and (3) the head vertex type.
  - Also here labels are assumed to be shorthand for a special property.
    ```

Graph-type cores

- The set of element types that defines the core of a graph type
  - So we ignore for the moment graph constraints and graph attributes.
- No explicit inheritance or subtype declarations are considered.

Our basic concrete syntax for examples:

- `(:Person { name STRING, birthdate DATE })`
- `(:City Place { name STRING, url URL })`
- ```
  (:Person { name STRING, birthdate DATE })
    -[:livesIn { start DATE}]->
  (:City Place { name STRING, url URL })
  ```
  ```

Strict semantics and non-strict semantics

- In the report we distinguish:
  - **Strict conformance**: no undeclared subtyping, focus on predictable size of instances
  - **Non-strict conformance** (or just **conformance**): reflects that instances of subtypes also conform, e.g., elements can have additional attributes and still conform.
- The current discussion focuses on **strict conformance**
  - But report also defines the associated non-strict conformance
- We assume predefined notions of strict conformance and conformance for attribute value types.

Matching of records and record types

- Fundamental notion for determining if element conforms to a type
- Based on matching of records and record types:
  - Consider record type `{street STRING, city STRING}`
  - `{street="Malet st", city="London"}` is an **exact match**
  - `{street="Malet st", city="London", country="UK"}` is an **over match**
  - `{street="Malet st"}` is an **under match**
  - `{street=5, city="London", country="UK"}` is **no match**.

Matching of vertex types

- Consider vertex type `(:Person {name STRING, birthdate DATE})`

- `(jan1: {Person=lbl, name="Jan Hidders", birthdate="1980-01-01"})`
  - is an **exact match**
- `(jan2: {Person=lbl, name="Jan Hidders", birthdate="1980-01-01", nationality="Dutch"})`
  - is an **over match**
- `(jan3: {Person=lbl, name="Jan Hidders"})`
  - is an **under match**
- `(jan4: {Person=lbl, name=43, birthdate="1980-01-01"})`
  - is **no match**

Matching of edge types

- Consider vertex type
  - ```
    (:Person { name STRING, birthdate DATE })
      -[:worksIn { start DATE}]->
    (:City { name STRING, url URL })
    ```
- For a vertex to be an **exact match** of a vertex type the following must hold:
  - The tail vertex must be an **exact match** of the tail type.

    ```
    (:Person { name STRING, birthdate DATE })
    ```
  - The content must be a an **exact match** of the content type.

    ```
    :worksIn { start DATE}
    ```
  - The head vertex must be an **exact match** of the head type.

    ```
    (:City { name STRING, url URL })
    ```
- Similarly for **under match** and **over match**.
    ```

Option #1: At-least-one-match

Strict-conformance definition

Definition #1: We say that a property graph **strictly conforms to** a graph-type core if for every element in the graph there is an element type in the core that it exactly matches.

Strict-conformance examples (1/4)

Graph-type core:

- `(:Person {name STRING, birthdate DATE})`
- `(:City {name STRING, url URL})`
- `(:Person {name STRING, birthdate DATE }) -[:worksIn { start DATE}]-> (:City {name STRING, url URL})`
- `(:Person {name STRING, birthdate DATE }) -[:livesIn { start DATE}]-> (:City {name STRING, url URL})`

Graph

```
(v1 :Person {name="Jan Hidders",
     birthdate="1980-01-01"})
(v2 :City {name="London",
           url="www.london.org"})
(v3 :City {name="Brussels",
           url="www.brussels.org"})
(v1)-[e1 :worksIn {start="2020-01-01"}]->(v2)
(v1)-[e2 :livesIn {start="2015-01-01"}]->(v3)
```

Strictly conforms.

Strict-conformance examples (2/4)

Graph-type core:

- `(:Person {name STRING, birthdate DATE})`
- `(:City {name STRING, url URL})`
- `(:Person {name STRING, birthdate DATE }) -[:worksIn { start DATE}]-> (:City {name STRING, url URL})`
- `(:Person {name STRING, birthdate DATE }) -[:livesIn { start DATE}]-> (:City {name STRING, url URL})`

Graph

```
(v1 :Person {name="Jan Hidders",
     birthdate="1980-01-01"})
(v2 :City {name="London",
           url="www.london.org"})
(v3 :City {name="Brussels",
           url="www.brussels.org"})
(v1)-[e1 :worksIn {start="2020-01-01"}]->(v2)
```

Strictly conforms

- **`livesIn`** is not instantiated, but not all types need to be

Strict-conformance examples (3/4)

Graph-type core:

- `(:Person {name STRING, birthdate DATE})`
- `(:City {name STRING, url URL})`
- `(:Person {name STRING, birthdate DATE }) -[:worksIn { start DATE}]-> (:City {name STRING, url URL})`
- `(:Person {name STRING, birthdate DATE }) -[:livesIn { start DATE}]-> (:City {name STRING, url URL})`

Graph

```
(v1 :Person {name="Jan Hidders",
             birthdate="1980-01-01",
             birthplace="Deventer"})
(v2 :City {name="London",
           url="www.london.org"})
(v3 :City {name="Brussels",
           url="www.brussels.org"})
(v1)-[e1 :worksIn {start="2020-01-01"}]->(v2)
(v1)-[e2 :livesIn {start="2015-01-01"}]->(v3)
```

Does not strictly conform:

- **v1** is over match

Strict-conformance examples (4/4)

Graph-type core:

- `(:Person {name STRING, birthdate DATE})`
- `(:City {name STRING, url URL})`
- `(:Person {name STRING, birthdate DATE }) -[:worksIn { start DATE}]-> (:City {name STRING, url URL})`
- `(:Person {name STRING, birthdate DATE }) -[:livesIn { start DATE}]-> (:City {name STRING, url URL})`

Graph

```
(v1 :Person {name="Jan Hidders"})
(v2 :City {name="London",
           url="www.london.org"})
(v3 :City {name="Brussels",
           url="www.brussels.org"})
(v1)-[e1 :worksIn {start="2020-01-01"}]->(v2)
(v1)-[e2 :livesIn {start="2015-01-01"}]->(v3)
```

Does not strictly conform:

- **v1** is under match

Option #2: Exactly-one-match

Strict-conformance definition

Definition #2: We say that a property graph **strictly conforms to** a graph-type core if for every element in the graph there is exactly one element type in the core that it exactly matches.

- Motivation:
  - Creates one-to-one correspondence between graph and graph-structure type: elements “live” in exactly one type

Strict-conformance example

Graph-type core:

- `$personWithLongName = (:Person {name VARCHAR(50)})`
- `$personWithShortName = (:Person {name VARCHAR(15)})`

Graph

- `(v1 :Person {name="Mary Anne Cunningham"}`
- `(v2 :Person {name="George Walsh"}`

Does not strictly conform:

- `v2` strictly conforms to both types

Is it really different?

- What is the difference with the **at-least-one-match semantics**?
- None, if every attribute value strictly conforms to exactly one attribute type
  - **NOTE:** is not the same as all values having a **unique most specific type**
- Opinion in GS-Basic varies on if this assumption is valid.
  - Why it might not hold:
    - VARCHAR(*n*), ARRAY(*n*), Union types, Optional attributes
  - Why it might hold:
    - In some cases we can normalize the core so that it does for the types in the graph-type core, e.g.,
      - union types (if not in collection types), optional attributes,
      - VARCHAR and ARRAY under certain assumptions

Option #3: Homomorphism-based

Intuitive motivation

- Consider a graph-type core:
  - `(:City { name STRING, url URL })`
  - `(:Person { name STRING, birthdate DATE })`
  - `(:Person { name STRING, birthdate DATE })`
    `-[:worksIn { start DATE}]->`
    `(:City { name STRING, url URL })`
- Assuming all vertex types in edge types are explicitly declared, this maps straightforwardly to a graph where types are the attribute values:
  - `(c :City { name=STRING, url=URL })`
  - `(p :Person { name=STRING, birthdate=DATE })`
  - `(p)-[w: worksIn { start=DATE}]->(c)`
- This implies a natural notion of homomorphism from property graphs to (property graphs that represent) graph-type cores.

Graph-to-Core homomorphisms

Core graph:

- `(c :City {name=STRING, url=URL})`
- `(p :Person {name=STRING, birthdate=DATE})`
- `(p)-[w1: worksIn {start=DATE}]->(c)`
- `(p)-[w2: livesIn {start=DATE}]->(c)`

Graph-to-Core Homomorphism:

A substitution of (1) vertices with vertices and (2) property values with a type they strictly conform to, that maps the property graph to the the schema graph.

Property graph:

- `(c1 :City {name="London", url="london.org.uk"})`
- `(c2 :City {name="Brussels", url="www.brussels.eu"})`
- `(p1 :Person {name="Jan Hidders", birthdate="01-01-1980"})`
- `(p1)-[w1: worksIn {start="15-01-2020"}]->(c1)`
- `(p1)-[w2: livesIn {start="15-09-2015"}]->(c2)`

E.g. { `c1` ↦ `c`, `"London"` ↦ `STRING`, `c2` ↦ `c`, … }

Strict-conformance definition

Definition #3: We say that a property graph **strictly conforms to** a graph-structure type if there is a graph-to-core homomorphism from the property graph to the graph of the graph-type core.

Is it really different? Is it a separate proposal?

- Is there actually a difference with the **at-least-one-match** and the **exactly-one-match** semantics?
- These all coincide with the **homomorphism-based semantics** if every attribute value strictly conforms to exactly one attribute type
- As stated earlier: opinion in GS-Basic is divided on if this assumption is reasonable
- If it is not valid, validation complexity is non-tractable where it is tractable for all other proposals.

Concern #1: Isolation-awareness

Motivating example

- Graph-type core:
  - `(:Person { name STRING, birthdate DATE })`
  - `(:Person { name STRING, birthdate DATE })`
    `-[:worksIn { start DATE}]->`
    `(:City { name STRING, url URL })`
- This, under the previous semantics does not allow **`worksIn`** edges.
  - Because there is no vertex type **`(:City { name STRING, url URL }`**
- This indicates room for a more sophisticated semantics:
  - The edge type justifies the nodes involved in the edge
  - The vertex type is only needed to justify isolated vertices.
- So under the given graph-structure type persons without work are allowed to exist but cities must have at least one person working in them.

More motivation

- In RDF the general rule is that nodes cannot exist without edges
  - This interpretation allows to enforce similar behavior for property graphs
- Permanently-isolated nodes of a given type are how you represent an SQL table in a GQL graph
- Conceptual data modelling methods such as Fact-Based Modeling, have a related notion: *independent object types*.

Strict-conformance definition

Definition #4: We say that a property graph **strictly conforms to** a graph-type core if

- for every edge in the graph there is an edge type in the core that it exactly matches and
- for every isolated vertex in the graph there is a vertex type in the core that it exactly matches.

NOTE: This is the isolation-aware variant of the at-least-one-match semantics. It is also possible to define such a variant of the exactly-one-match semantics and the homomorphism-based semantics.

Strict-conformance examples (1/3)

Graph-type core:

- `(:Person { name STRING, birthdate DATE })`
- `(:Person { name STRING, birthdate DATE }) -[:worksIn { start DATE}]-> (:City { name STRING, url URL })`

Graph

- `(v1 :Person {name="Jan Hidders", birthdate="1980-01-01"})`
- `(v2 :City {name="London", url="www.london.org"})`
- `(v1) -[e1 :worksIn {start="2020-01-01"}]-> (v2)`

Does strictly conform

Strict-conformance examples (2/3)

Graph-type core:

- ```
  (:Person { name STRING,
             birthdate DATE })
  ```
- ```
  (:Person { name STRING,
             birthdate DATE })
    -[:worksIn { start DATE}]->
  (:City { name STRING,
           url URL })
  ```

Graph

- ```
  (v1 :Person {name="Jan Hidders",
               birthdate="1980-01-01"})
  ```

Does strictly conform
  ```

Strict-conformance examples (3/3)

Graph-type core:

- `(:Person { name STRING, birthdate DATE })`
- `(:Person { name STRING, birthdate DATE }) -[:worksIn { start DATE}]-> (:City { name STRING, url URL })`

Graph

- `(v2 :City {name="London", url="www.london.org"})`

Does not strictly conform:

- **v2** is isolated, but does not match vertex type

Should the semantics be isolation-aware?

- **GOOD:** Gives meaning to otherwise meaningless graph-structure types
- **GOOD:** Adds useful concept
- **BAD:** Adds some complexity to semantics
- **GOOD/BAD:** Concept can be expressed as well with (sophisticated) cardinality constraints

Open Questions

Open questions / future work

- How to define the semantics of optional attributes?
- How to define the semantics of union types?
- What is an appropriate syntax for type variables?
- What is the syntax and semantics for inheritance?
- How to introduce (partially) open types for partially open schemas?
- Which constructs can approximate conceptual data models?
- What is the semantics of undirected-edge types?
- Why and how to introduce nominal typing?

Conclusion

Conclusion

The issues that keep us from making a single clear recommendation:

- Can we assume that every attribute value has exactly one attribute type it strictly conforms to?

- Do we want the semantics to be isolation-aware?

Agenda

- Goal
- Background
- Motivating use cases
- Requirements distilled
- Proposed CRUD behaviors
- Formal definitions
- Work in progress

MMX-073r1
LDBC OAEP-2023-04 Document A2 of A3 (7 of 8)
“Meta-properties overview”

Meta-properties

WG3 report: 2020-6-4

Bei Li

Goal

- Share the current status of the subtrack
- Gather feedback
- Syntaxes are for illustration purposes only

Background: property graph description

A property graph is defined as consisting of:

(1) a set of vertices such that each vertex has some associated vertex content,

(2) a set of edges such that each edge has an associated (a) tail vertex, (b) edge content and (c) head vertex. The tail and head vertices of each edge must be in the set of vertices.

The content that is associated with vertices and edges is a finite record that maps attribute names to attribute values.

Both vertices and edges are assumed to be represented by an abstract identity, and so it is possible that a graph contains two vertices with the same content, and two edges that have the same tail vertex, head vertex and content.

* definition adopted from ISO/IEC JTC1 SC32 WG3 WG3:mmx069-LDBC_PGS-B_BAS-01

Background: property graph definition

Definition: A property graph is a tuple $G = (V, E, A, \varrho, \alpha)$ where

- V is a set of vertex identities,
- E is a set of edge identities,
- A is a set of attribute identities such that V, E and A are pairwise disjoint,
- $\varrho : E \rightarrow (V \times V)$ maps each edge identity to a pair of vertex identities
- $\alpha : A \rightarrow ((V \cup E) \times \mathcal{A} \times \mathcal{V})$ maps each attribute identity to a triple containing (1) the element that it is an attribute of, (2) its name and (3) its value. Every attribute identity in A is identified by the first two components.

Motivating use case: Wikidata

qualifier to constrain the validity of the value(s)

Berlin (Q64) (item) Node

population (P1082) (property) → 3,500,000 (property value) Property

point in time (P585) (qualifier) → 2005 (qualifier value)

determination method (P459) (qualifier) → estimation process (Q791801) (qualifier value)

Meta-property

Cannot be modelled natively with Property Graph

Motivating use case: others

- Provenance
- Fuzzy and trust
- Valid time
- Timestamp
- Units
- Location

Requirements: must-haves

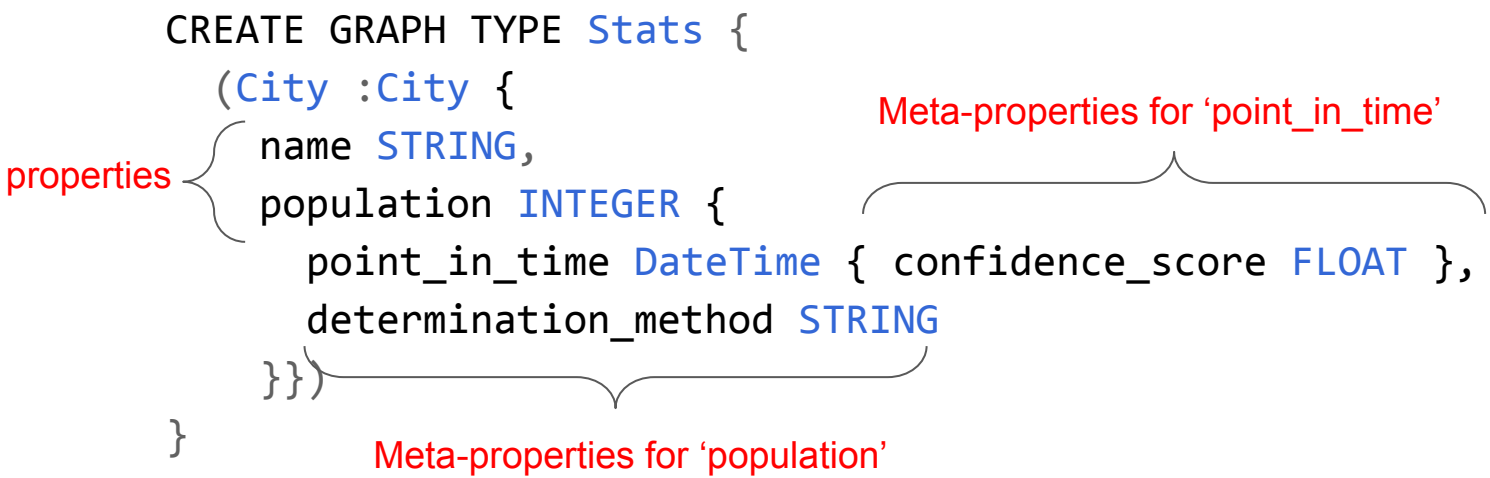
1. Any extensions to the PG data model should not break existing work.
2. Support for meta-properties of simple or complex-valued properties.
3. Meta-properties should be queryable similar to regular properties.
4. The typing system should be able to control which meta-properties are allowed where and what their values are. This applies both at the graph schema level and within the property values.

Requirements: optional

5. Support for meta-properties of components of complex values (e.g. record attributes, list elements, map keys or map values).
    a. For example, an Address property which is broken down to street number, name and city could have meta-properties at the level of the Address record as a whole, but also at the level of each constituent components.

Simple value types

DDL


CREATE GRAPH TYPE Stats {
(City :City {
name STRING,
population INTEGER {
point_in_time DateTime { confidence_score FLOAT },
determination_method STRING
}})
}
properties
Meta-properties for ‘point_in_time’
Meta-properties for ‘population’

DML: INSERT

```
INSERT (n:City
        {
          name: "Berlin",
          population: 3.5M {
            point_in_time: 2005 { confidence_score: 0.99 }
          }
        })
RETURN n
```

DQL: READ

```
MATCH (n:City { name = "Berlin" })
RETURN n.population, n.population.point_in_time,
       n.population.*
```

Binding table

population	population.point_in_time	population.*
3.5M	2005	3.5M { point_in_time … }

DML: UPDATE

```
UPDATE (n:City { name = "Berlin" })
SET n.population = 3.5M { point_in_time: 2005 { confidence_score: 1.0 } }
```

```
UPDATE (n:City { name = "Berlin" })
SET n.population.point_in_time = 2005 { confidence_score: 1.0 }
```

```
UPDATE (n:City { name = "Berlin" })
SET n.population.point_in_time.confidence_score = 1.0
```

DML: DELETE

```
MATCH (n:City { name = "Berlin" })
REMOVE n.population

MATCH (n:City { name = "Berlin" })
REMOVE n.population.point_in_time

MATCH (n:City { name = "Berlin" })
REMOVE n.population.point_in_time.confidence_score
```

Complex value types

DDL: collection type

```
CREATE GRAPH TYPE Stats {
  (City :City {
    name STRING,
    zip_codes [INTEGER { creation_time DateTime }] {
      point_in_time DateTime
    }})
}
```

Meta-properties for individual zip codes

Meta-properties for ‘zip_codes’

DML: INSERT - collection type

```
INSERT (n:City
        {
          name: "Berlin",
          zip_codes: [10115 { creation_time: 1962 },
                      10117 { creation_time: 1965 }] {
            point_in_time: 2005
          }
        })
RETURN n
```

DDL - record type

```
CREATE GRAPH TYPE Stats {
  (City :City {
    name STRING,
    mayor {
      first_name STRING { last_edit UINT64 },
      last_name  STRING { last_edit UINT64 }
    } { since DateTime })
}
```

Meta-properties for first / last name

Meta-properties for ‘major’

DML: INSERT - record type

```
INSERT (n:City {
          name: "Berlin",
          mayor: {
            first_name: "Michael" { last_edit: 1420070400 } ,
            last_name:  "Müller"  { last_edit: 1420070400 } } {
              since: 12/11/2014
            }
          })
RETURN n
```

Formal definitions

Formal definition: perspective 1 - record type

Definition: The set of **annotated attribute types** $\mathcal{T}_{attr}^{@}$ is recursively defined as a type of the form $type_1@type_2$ where:

- $type_1$ is one of the following :
  - a basic type
  - a record type with fields that have all annotated attribute types
  - a collection type parameterized with annotated attribute types
- $type_2$ is a record type where all field types are again of annotated attribute types.

Major changes: (a) introduce a new data type “annotated attribute type”. The definition of property graph remains unchanged.

Examples: perspective 1

Type	Example values
STRING STRING@{}	“Berlin”
INTEGER @{ point_in_time: DateTime }	2.5M 2.5M { point_in_time: 2005 }

Examples: perspective 1 cont.

Type	Example values
[INTEGER@{ creation_time: DateTime }] @{ point_in_time: DateTime }	[10115 { creation_time: 1962 }, 10117 { creation_time: 1965 }] { point_in_time: 2005 } [10115 { creation_time: 1962 }, 10117 { creation_time: 1965 }] [10115 { creation_time: 1962 }, 10117] [10115, 10117] null { point_in_time: 2005 }
{ first_name: STRING } @{ since: DateTime }	{ first_name: “Michael” } { since: 12/2004 } { first_name: “Michael” } null@{ since: 12/2004 }

Formal definition: perspective 2 - APG

Definition: A property graph with meta-properties is a tuple $G = (V, E, A, \varrho, \alpha)$ where

- V is a set of vertex identities,
- E is a set of edge identities,
- A is a set of attribute identities such that V, E and A are pairwise disjoint,
- $\varrho : E \rightarrow (V \times V)$ maps each edge identity to a pair of vertex identities
- $\alpha : A \rightarrow ((V \cup E \cup \mathbf{A}) \times \mathcal{a} \times \mathcal{V})$ maps each attribute identity to a triple containing (a) the element that it is an attribute of, (b) its name and (c) its value, so that an attribute cannot directly or indirectly be its own attribute. ~~Every attribute identity in A is identified by the first two components.~~

Major changes: (a) We in addition allow elements of A in the first component of α, (b) We drop the identification constraint for attributes.

Examples: perspective 2

Identity	the element that it is an attribute of	name	value
`$nn`	`$n`	`name`	`“Berlin”`
`$np`	`$n`	`population`	`3.5M`
`$np1`	`$np`	`point_in_time`	`2005`
`$np2`	`$np`	`provenance`	`"wikipedia"`
`$np11`	`$np1`	`writer`	`"admin"`

In this example, $n is a node variable.

Perspective 2 - observation

- Pros
  - Get multivalued properties for free, because each property has an identity, thus properties with the same name on the same graph element can coexist.
  - Get meta-properties on the elements of the collections for free (using multi-valued properties)
- Cons
  - Need to extend the graph elements definition in Property Graph, to include properties in addition to nodes and edges.
  - Cannot attach meta-properties to components of complex values

Comparison: summary

Requirements	Record type	APG
1, Any extensions to the PG data model should not break existing work.	Yes	Yes
2, Support for meta-properties of simple or complex values.	Yes	Yes
3, Meta-properties should be queryable just like regular properties.	Yes	Yes
4, The typing system should be able to control which meta-properties are allowed where and what their values are.	Yes	Yes
5, Support for meta-properties of components of complex values	**Yes**	**No**

Work in progress

Ambiguity

```
CREATE GRAPH TYPE Stats {
  (City :City {
    score: {
      livability FLOAT,     // livability score of the city
      confidence FLOAT      // confidence about the city
    } {
      confidence FLOAT      // confidence about the `score`
    }})
}
```

Which ‘confidence’ does ‘n.score.confidence’ refer to?

Disambiguation: option 1

Distinguish the two cases:

- “.” to access values
- “@” to access meta-properties
- “#” to access both values and their meta-properties

```
n.score.confidence  // ‘confidence’ field of the value.
n.score@confidence  // ‘confidence’ meta-property.
n.score#            // the entire value and its meta-properties.
```

Disambiguation: option 1 examples

```
mayor       // returns the plain value of ‘major’.
{ first_name: "Michael", last_name:  "Müller" }


mayor.*     // returns everything about ‘major’’s value.
{ first_name: "Michael" { last_edit: 1420070400 } ,
  last_name:  "Müller"  { last_edit: 1420070400 } }


mayor@*     // returns everything about ‘mayor’’s meta-properties.
{ since: 12/11/2014 }

mayor#*     // returns everything about ‘mayor’. ‘*’ is optional.
{ first_name: "Michael" { last_edit: 1420070400 } ,
  last_name:  "Müller"  { last_edit: 1420070400 } } { since: 12/11/2014 }
```

Disambiguation: option 2

- Allow meta-properties on atomic values and collections.
- Meta-properties on records are represented as fields in the record prefixed with @.

Comparison option 1 and option 2

Option 1

```
X = {
 name: "Berlin"@{src: "Wikipedia"},
 population: 3.5M@{precision: .1M},
 mayor: {
   first_name: "Michael"
          @{verified: true},
   last_name: "Müller"
          @{verified: false}
 }@{since: 4-May-2005}
}
```

- X.name ⇒ "Berlin"
- X.name@src ⇒ "Wikipedia"
- X.mayor.first_name ⇒ "Michael"
- X.mayor@since ⇒ 4-May-2005

Option 2

```
X = {
 name: "Berlin"@{src: "Wikipedia"},
 population: 3.5M@{precision: .1M},
 mayor: {
   first_name: "Michael"
          @{verified: true},
   last_name: "Müller"
          @{verified: false},
   @since: 4-May-2005
 }
```

- X.name ⇒ "Berlin"
- X.name.src ⇒ "Wikipedia"
- X.mayor.first_name ⇒ "Michael"
- X.mayor.@since ⇒ 4-May-2005

Comparison option 1 and option 2

Option 1

- ```
  X = {
    name: "Berlin"@{src: "Wikipedia"},
    population: 3.5M@{precision: .1M},
    mayor: {
      first_name: "Michael"
              @{verified: true},
      last_name: "Müller"
              @{verified: false}
    }@{since: 4-May-2005}
  }
  ```
- ```
  X.mayor.* ⇒ {first_name: "Michael"
                @{verified: true},
               last_name: "Müller"
                @{verified: false}
              }
  ```

Option 2

- ```
  X = {
    name: "Berlin"@{src: "Wikipedia"},
    population: 3.5M@{precision: .1M},
    mayor: {
      first_name: "Michael"
              @{verified: true},
      last_name: "Müller"
              @{verified: false},
      @since: 4-May-2005
    }
  ```
- ```
  X.mayor.* ⇒ {first_name: "Michael"
                @{verified: true},
               last_name: "Müller"
                @{verified: false}
              }
  ```

Comparison option 1 and option 2

Option 1

- ```
  X = {
    name: "Berlin"@{src: "Wikipedia"},
    population: 3.5M@{precision: .1M},
    mayor: {
      first_name: "Michael"
              @{verified: true},
      last_name: "Müller"
              @{verified: false}
    }@{since: 4-May-2005}
  }
  ```
- `X.mayor@*` ⇒ `{since: 4-May-2005}`

Option 2

- ```
  X = {
    name: "Berlin"@{src: "Wikipedia"},
    population: 3.5M@{precision: .1M},
    mayor: {
      first_name: "Michael"
              @{verified: true},
      last_name: "Müller"
              @{verified: false},
      @since: 4-May-2005
    }
  ```
- `X.mayor.@*` ⇒ `{since: 4-May-2005}`

Questions?

WG3:MMX-074
LDBC OAEP-2023-04 Document A3 of A3 (7 of 8)
“Key Constraints”

Key Constraints

15 June 2020

LDBC PGSWG

Design principles for key constraints

1. Should support definition of key constraints on *node*, *edge*, and *path* objects (but, not critically rely on this initial scoping).
2. Key constraints are a variety of equality-generating dependencies (EGDs), of which, in the relational world, functional dependencies (and hence super/candidate/primary keys) are a special case.
3. Basic ingredients for defining key constraints: object labels (aka “types”), property-values, and object identities.
4. We adopt homomorphic matching semantics (instead of isomorphic matching semantics) for pattern matching component of key constraints, as this matches user intuition and is computationally well-behaved.
5. We balance a visionary design for the future of property graph key constraints, with a design with good potential for practical impact.

Key constraints

A key constraint is an expression of the form

$$\{p_1, \ldots, p_n\} : \{s_1, \ldots, s_m\} \rightarrow \mathbf{x}$$

where:

- **x**, the target of the constraint, is of the form *x* or of the form *x.a*, where *x* is an object variable and *a* is an attribute name,
- $\{s_1,\ldots,s_m\}$ is a set of $m \geq 0$ selectors on objects *v* and their attributes *v.a*, and
- $\{p_1,\ldots,p_n\}$ is a graph pattern
  - non-empty set of atomic expressions, i.e., expressions of the form: *t(y)* [type patterns], *(y, z, w)* [edge patterns], *(y, (z, r), w)* [path patterns], *y.a θ z.b,* or *y.a θ c* [attribute patterns]
  - essentially, a Conjunctive Regular Path Query with attribute filters

such that every variable appearing in **x** and $\{s_1,\ldots,s_m\}$ also appears in $\{p_1,\ldots,p_n\}$.

Key constraints

Intuitively, a key constraint

$$\{p_1, \ldots, p_n\} : \{s_1, \ldots, s_m\} \rightarrow \mathbf{x}$$

states that "In the graph query defined by $\{p_1,\ldots,p_n\}$,
x is identified by

- the identities of objects and
- values of attributes

selected by $\{s_1,\ldots,s_m\}$."

See report for full details, including formal syntax/semantics.

Key constraints

In other words, the key constraint

$$\{p_1, \ldots, p_n\} : \{s_1, \ldots, s_m\} \rightarrow \mathbf{x}$$

holds on a property graph G if and only if,

for any two matches m_1 and m_2 of $\{p_1,\ldots,p_n\}$ in G,

if $m_1(s) = m_2(s)$ for every element s of $\{s_1,\ldots,s_m\}$, then $m_1(\mathbf{x}) = m_2(\mathbf{x})$.

See report for full details, including formal syntax/semantics.

Key constraints

We will write

$$\{p_1, \ldots, p_n\} : \{s_1, \ldots, s_m\} \rightarrow \mathbf{x}$$

in pseudocode as

```
WHERE p_1, ..., p_n
REQUIRE s_1, ..., s_m IDENTIFIES x
```

Key constraints: node key example 1

[Key constraint for nodes, only reasoning about local attributes]

The “name” property should be a key of countries. More precisely, the name property of a country node should identify the country node. Here, we have the graph pattern { Country(*x*) }, selector set { *x*.name }, and target *x*.

```
WHERE (x:Country)
REQUIRE x.name IDENTIFIES x
```

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 $n_1 = n_2$.

Key constraints: node key example 2

[Key constraint for nodes, reasoning about both local attributes and graph topology]
Cities are identified by their name and the country they are in. 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.

```
WHERE (x:City)-[y:isPartOf]->(z:Country)
REQUIRE x.name, z IDENTIFIES x
```

In other words, 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 $n_1 = n_2$.

Key constraints: edge key example

[Key constraint for edges]

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.

```
WHERE (x:Person)-[y:studyAt]->(z:University)
REQUIRE x, y.year, z IDENTIFIES y
```

In other words, given any two edges e_1 and e_2 labeled `studyAt`, 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 `year`, then it must be the case that $e_1 = e_2$.

Key constraints: path key example

[Key constraint for paths] Suppose that companies directly and indirectly transfer money between each other (initiated by a transfer activity), and it is crucial (e.g., for auditing) that chains of transfer between companies are unique. For a given transfer, the `transferID` property value of each edge in the transfer chain must be the same as the `ID` property value of the transfer activity.

```
WHERE (a:Activity)-[x:initiatedBy]->(c1:Company)-/v <([:Transfer {transferID: a.ID} ])*> /->(c2:Company)
REQUIRE c1, c2 IDENTIFIES v
```

In other words, given any two paths v_1 and v_2 , if they both have the same `Company` source node, the same `Company` target node, every edge along both paths has label `Transfer` and have `transferID` value equal to the `ID` value of an `Activity` node having an `initiatedBy` edge to the source node, then it must be the case that $v_1 = v_2$.

Questions

1. What is the minimally expressive fragment necessary to express schemas that capture E/R models or to express constraints on data imported from a relational database?

2. Are key constraints computationally well-behaved?

Can we enforce relational keys as PG key constraints?

Yes.

The proposed model can exactly describe constraints that arise from translating relational primary key constraints (using natural relational-to-graph translations).

- Simple fragment, see report for details.

If foreign key constraints are also present, then we need to constrain cardinalities (need to be able to say “exactly one”).

- Cardinality constraints are currently the main focus of our work in the GS keys/constraints group.

Are key constraints computationally well-behaved?

Yes.

We have established that the validation problem (i.e., determining whether or not a given property graph satisfies a given key constraint), while intractable in general, is in polynomial time for a large subclass of CRPQ which covers over 99% of graph patterns observed in practice, namely:

tree-shaped graph patterns

Furthermore, our semantics of keys allows us to use existing query evaluation infrastructure for the validation problem

A subgroup is continuing this study, also for problems such as incremental validation, error reporting, and implication.

Open design issues

$$\{p_1, \ldots, p_n\} : \{s_1, \ldots, s_m\} \rightarrow \mathbf{x}$$

I. We do not place any constraints on the structure of the graph query defined by $\{p_1,\ldots,p_n\}$. It is natural to consider restrictions such as acyclicity, bounded size, connectivity, ….

   Similarly, we could constrain $\{s_1,\ldots,s_m\}$, e.g., no edge or path variables.

II. We could consider richer/weaker query constructs or even completely different query languages for expressing queries.

III. Further basic reasoning problems should be studied, such as incremental validation, consistency, and implication.

IV. Better practical syntax and/or graphical notation for keys (along with the general graphical notation for schemas) should be studied.