A Researcher's Digest of GQL

Nadime Francis ✉
Laboratoire d'Informatique Gaspard Monge,
Université Gustave Eiffel, CNRS, France

Amélie Gheerbrant ✉
IRIF, Université Paris Cité, CNRS,
Paris, France

Paolo Guagliardo ✉
School of Informatics,
University of Edinburgh, UK

Leonid Libkin ✉
University of Edinburgh, UK
RelationalAI, France
ENS, PSL University, France

Victor Marsault ✉
Laboratoire d'Informatique Gaspard Monge,
Université Gustave Eiffel, CNRS, France

Wim Martens ✉
Universität Bayreuth, Germany

Filip Murlak ✉
University of Warsaw, Poland

Liat Peterfreund ✉
Laboratoire d'Informatique Gaspard Monge,
Université Gustave Eiffel, CNRS, France

Alexandra Rogova ✉
IRIF, Université Paris Cité, CNRS, Paris, France
Data Intelligence Institute of Paris, Inria

Domagoj Vrgoč ✉
University of Zagreb, Coratia
Pontificia Universidad Católica de Chile,
Santiago, Chile

Abstract

GQL (Graph Query Language) is being developed as a new ISO standard for graph query languages to play the same role for graph databases as SQL plays for relational. In parallel, an extension of SQL for querying property graphs, SQL/PGQ, is added to the SQL standard; it shares the graph pattern matching functionality with GQL. Both standards (not yet published) are hard-to-understand specifications of hundreds of pages. The goal of this paper is to present a digest of the language that is easy for the research community to understand, and thus to initiate research on these future standards for querying graphs. The paper concentrates on pattern matching features shared by GQL and SQL/PGQ, as well as querying facilities of GQL.

2012 ACM Subject Classification Theory of computation → Database theory; Theory of computation → Database query languages (principles); Information systems → Graph-based database models; Information systems → Structured Query Language

Keywords and phrases GQL, Property Graph, Query Language, Graph Database, Pattern matching, Multi-Graph

Digital Object Identifier 10.4230/LIPIcs.ICDT.2023.1

Category Invited Talk

Funding This work is supported by: a Leverhulme Trust Research Fellowship; EPSRC grant S003800; French ANR projects grants ANR-18-CE40-0031 (*QUID*), ANR-19-CE48-0019 (*EQUUS*), and ANR-21-CE48-0015 (*Verigraph*); German Research Foundation (DFG) projects 431183758 and 369116833; ANID Millennium Science Initiative Program, Code ICN17_002; ANID Fondecyt Regular project 1221799; NCN grant 2018/30/E/ST6/00042.

Acknowledgements The authors are grateful to members of the ISO/IEC JTC1 SC32 WG3 committee and especially Fred Zemke for many comments on our formalization of the language.

© Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč;
licensed under Creative Commons License CC-BY 4.0

26th International Conference on Database Theory (ICDT 2023).
Editors: Floris Geerts and Brecht Vandevoort; Article No. 1; pp. 1:1–1:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Graph databases have grown steadily in popularity this century. They handle data as it is viewed conceptually, making them easily applicable in many tasks where traditional relational databases are not easy or natural to use. While many early applications cited social networks and the Semantic Web as the key motivation (since in both cases data is naturally viewed as a graph), industry scale applications are much more diverse and include fraud detection, network management, medical data management, knowledge management, and even investigative journalism. There are several dozen graph database products on the market, including the current leader Neo4j, as well as both established and upcoming companies offering graph products (e.g., Oracle, Amazon, IBM, SAP, Redis, DataStax, TigerGraph, Memgraph, etc.).

Graph databases' widespread use happened without them having their lingua franca, which is the role that SQL is playing for relational databases. The landscape of graph languages – at least at first sight – is very varied. Neo4j has its own language called Cypher [18], which is also implemented in other products, including SAP HANA and Amazon Neptune. Oracle introduced its language PGQL [34]; TigerGraph has GSQL [13], and several products use the non-declarative graph traversal language Gremlin [33]. However, upon a closer examination, one discovers that declarative languages are more like different dialects of the same language rather than different languages altogether. This led to a proposal to define a new unifying standard for a Graph Query Language (GQL) [36]. The proposal was given a go-ahead in 2019, and since then was taken up by the same committee that produces and maintains the SQL Standard. It is known as ISO/IEC JTC1 SC32 WG3 within the International Organization for Standardization, or ISO.

In fact, this committee develops two projects in parallel:

- SQL/PGQ, a new Part 16 of the SQL Standard, that defines querying graphs specified as views over a relational schema; it is expected to be published roughly at the time of the EDBT/ICDT 2023 conference.
- GQL, a standalone language for querying property graphs, that is expected to be published in late 2023 or early 2024.

The language of the Standard, even when published (behind paywall) is hardly of the kind that the research community is accustomed to. It consists of a grammar for the constructs, supplemented with syntax and semantic rules, the latter written in natural language describing an algorithm for computing the result of a particular operation (essentially a mix of prose and pseudocode). Such descriptions are long, far from formal definitions suitable for initiating research in the area, and often prone to misinterpretation. To researchers, such a text is therefore much like a 500+ page legal document, instead of a workable definition that helps them understand the essence of the language.

This motivates the goal of the present paper: *to distill, in a form accessible to the database research community, the principal elements of the forthcoming GQL Standard, and provide their formal semantics.*

The idea of finding calculi underlying programming languages and providing their formal semantics is mainstream in the programming languages field. Recently we saw it extended to database query languages, specifically to core fragments of SQL [10, 22, 7] and Cypher [18]. The present paper follows this trend. It provides a significant simplification of the GQL Standard, which at the same time covers its key features, and yet is sufficiently simple to provide its formal semantics, thereby enabling its further study and opening up new avenues of research on graph query languages.

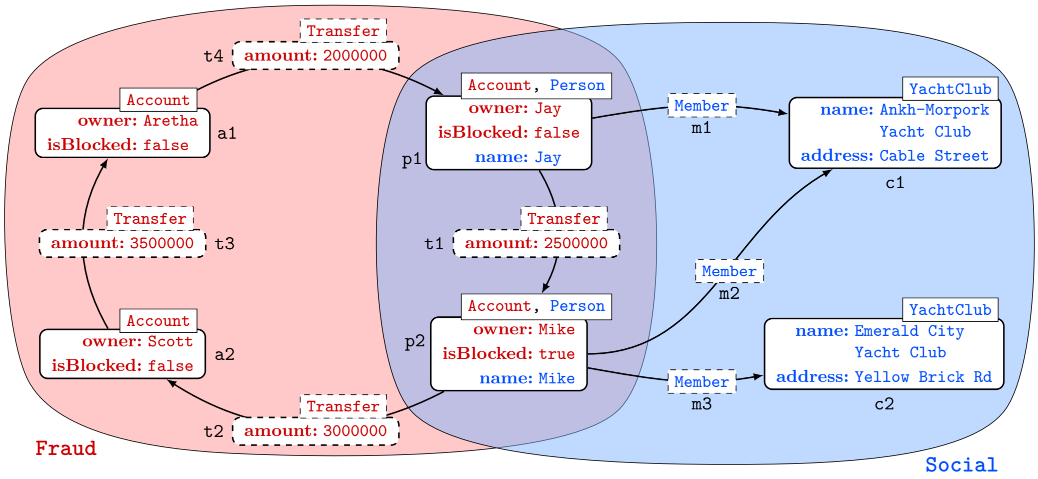

Figure 1 A database with graphs `Fraud` and `Social`.

We do not follow GQL letter to letter, for two reasons. Firstly, the Standard itself is not yet finalized, and what is written today may still change before it is published. Second, we choose to simplify some of the idiosyncrasies of a real-life language to better highlight its essential features. Queries presented here are close to the eventual features of the language – even if they change somewhat in the meantime. They come with a formal grammar that is a fragment of GQL's grammar, and a formal semantics, that is suitable as a starting point of new research in graph query languages. The paper focuses on *read-only GQL queries*, to which we will simply refer as GQL queries. That is, we do not yet consider data updates.

Previous Academic Work on GQL

The two graph languages currently standardized – GQL and SQL/PGQ – share their *pattern matching* facilities, which constitute the key part of any graph language. These were described in [12], by a group that included members of ISO's Standard group, as well as members of LDBC's Formal Semantics Working Group (FSWG), whose goal was to analyze and formalize the design of the language. FSWG then produced a theoretical reconstruction of the GQL and PGQ pattern language [16]. This paper is the next installment in the effort to distill PGQ and GQL standards for the research community.

Apart from this recent work on GQL, we note that academic foundations already influenced its design process. As seen in GQL's influence graph [19], the language draws inspiration from regular path queries [11, 30], STRUQL [14], GXPath [27], and regular queries [32].

2 GQL by Example

In this section we give a high-level description of GQL queries and their evaluation. The graph database model used by GQL is simply a collection of one or more property graphs. As an illustration, Figure 1 is a graph database consisting of two property graphs: the `Fraud` graph has information about bank transactions that are to be investigated for fraud, and the `Social` graph has information about people's social activities such as membership in a yacht club. Notice that these two graphs have a non-empty intersection: the nodes for Jay and Mike belong to both graphs, but they are seen in a different way and therefore have different labels and properties. In `Fraud`, the nodes have label `Account` and properties `owner` and `isBlocked`, indicating the status of the account. In `Social`, these nodes have label `Person` and property `name`.

We start with a simple query that looks for large (over $1M) transfers into a blocked account, and reports owners of accounts involved in such transfers:

```
1. USE Fraud
2. MATCH (x) -[z:Transfer WHERE z.amount>1000000]-> (y WHERE y.isBlocked=true)
3. RETURN x.owner AS sender, y.owner AS recipient
```

The reader familiar with Cypher will parse this query easily; it roughly follows Cypher's ascii-art syntax for expressing patterns, and also permits checking conditions on properties inside patterns. Basically, the pattern in line 2, namely:

```
(x) -[z:Transfer WHERE z.amount>1000000]-> (y WHERE y.isBlocked=true)
```

asks for nodes `x` and `y` that are connected with an edge `z` that is labeled with `Transfer`. Furthermore, the `amount` property of `z` should exceed one million and the `isBlocked` property of y should be `true`. Such patterns, called *path patterns* in GQL, are the main building block of GQL queries, and they roughly correspond to regular path queries (RPQs), which have been well studied in the research literature [30].

Note also that the query is preceded by a `USE` clause stating explicitly in which graph matches are sought. When evaluating a query, GQL keeps track of

- the *working graph*, which is the current graph in the database on which we do pattern matching and
- the *working table*, which contains intermediate results of the query, up to the current evaluation point.

Intuitively, the working table is a collection of records that gets passed from one part of the query to another in order to compute the final result. Thus, while GQL is a graph query language, it uses tables to represent intermediate and end-results of queries. In Section 4, we also discuss a third ingredient that GQL keeps track of, namely the *working record*.

Coming back to our sample query, in the first line we write `USE Fraud`, which turns the `Fraud` graph into our working graph. In line 2, we have our path pattern, preceded by the keyword `MATCH`. This clause is the main workhorse of GQL, and it tells us to do the matching of the pattern onto the working graph. When evaluating our query over the database from Figure 1, after executing line 2 of the query, we will be left with the following working table:

x	y	z
p1	p2	t1

Continuing in line 3, the working table is modified by keeping only the `owner` attribute of the nodes `x` and `y`, while renaming them, and the following is returned to the user:

sender	recipient
Jay	Mike

(1)

We next extend this query by checking for such transfers where both account owners are members of the same yacht club, reporting this time the address for the yacht club to send investigators to.

```
 1. USE Fraud {
 2.  MATCH (x) -[z:Transfer WHERE z.amount>1000000]-> (y WHERE y.isBlocked=true)
 3.  RETURN x.owner AS sender, y.owner AS recipient
 4.  THEN
 5.  USE Social
 6.  MATCH (x1) -[:Member]-> (z1:YachtClub) ,
 7.        (y1) -[:Member]-> (z1:YachtClub)
 8.  FILTER sender=x1.name AND recipient=y1.name
 9.  RETURN z1.address AS clubAddress
10. }
```

Here lines 1–3 repeat the previous query. The keyword `THEN` is used to pipe the result of this query to the following subquery. While the curly braces extend the scope of `USE Fraud` beyond `THEN`, in line 5 we switch the working graph to `Social` in order to match the pattern:

```
(x1) -[:Member]-> (z1:YachtClub) ,  (y1) -[:Member]-> (z1:YachtClub)
```

This pattern consists of two path patterns, separated by a comma. In GQL, the comma performs a join on the results of the two path patterns. From a theoretical point of view, it brings us in the realm of *conjunctive (two-way) regular path queries*. In GQL, such patterns are called *graph patterns*. When this pattern is evaluated over the `Social` graph, we obtain the following (fresh) working table:

x1	y1	z1
p1	p2	c1
p1	p1	c1
p2	p2	c1

(2)

this time with variables `x1`, `y1`, and `z1`. After evaluating the pattern, the `MATCH` statement makes the natural join of table (2) with table (1), leading to

sender	recipient	x1	y1	z1
Jay	Mike	p1	p2	c1
Jay	Mike	p1	p1	c1
Jay	Mike	p2	p2	c1

In this case, this will be the Cartesian product since the two working tables have no variables in common. The `FILTER` condition in line 8 selects only the first row of the latter table. The `RETURN` statement in line 9 tells us to keep only the `address` attribute of `z1`, renamed as `clubAddress`, resulting in:

clubAddress
Cable Street

This is also where our query ends, and the working table contains all the results to our query.

The examples we have seen thus far illustrate only a limited part of GQL since their variables only bind to single nodes or edges. Next, we show what happens to variables that can bind to *lists* and *paths*. Concerning lists, a query[1] such as

```
USE Fraud
MATCH TRAIL (x) ((y)-[:Transfer]->()){1,} (x)
RETURN x AS source, y AS moneyTrail
```

would return the following table.

[1] Notice that the query uses Cypher's ascii-art ( ) for nodes in the subexpressions `(x)`, `(y)`, and `()`, but also uses ( ) for indicating the subexpression over which `{1,}` is applied.

source	moneyTrail
p1	list(p1, p2, a2, a1)
p2	list(p2, a2, a1, p1)
a2	list(a2, a1, p1, p2)
a1	list(a1, p1, p2, a2)

Here, the variable `y` is bound to a *list of nodes*. The four outputs all describe the same trail, which is the only Transfer-cycle in the graph, but the bindings use different start nodes for `x` and therefore also order the nodes in the lists for `y` differently. Concerning paths, the query

```
USE Fraud
MATCH TRAIL p = (x) (-[:Transfer]->()){1,} (x)
RETURN x AS source, p AS path
```

would return the following table.

source	path
p1	path(p1, t1, p2, t2, a2, t3, a1, t4, p1)
p2	path(p2, t2, a2, t3, a1, t4, p1, t1, p2)
a2	path(a2, t3, a1, t4, p1, t1, p2, t2, a2)
a1	path(a1, t4, p1, t1, p2, t2, a2, t3, a1)

The output is similar to the output of the previous example, but this time we have the entire path instead of the list of nodes in each answer. We note that property graphs can have multiple edges with the same end-nodes, so the list of nodes in a path is not sufficient to determine the path.

3 Syntax of GQL

The full syntax of *GQL queries* is given in Figure 2 with $\mathbb{G}$ a set of property graphs, and the following pairwise disjoint countable sets: $\mathcal{L}$ of labels, $\mathcal{K}$ of keys, Const of value constants with a designated value null, and Vars of variables.

While somewhat intimidating at a first glance, the grammar can be roughly divided into four parts:

- *path patterns*, which mimic regular path queries [29, 30], but have additional features such as two-way navigation and conditioning;
- *graph patterns*, which generalize conjunctive two-way regular path queries [8] with the ability to return different types of paths;
- *queries*, which allow us to manipulate the results of graph patterns and combine their evaluation over different graphs in the database; and
- *expressions and conditions*, which allow filtering results obtained in previous three parts of GQL.

Of course, each of these parts has many specific features. For instance, path patterns allow using descriptors, which bind a node/edge to a variable, test its label or more complex conditions (e.g. `amount` is greater than 1000000). Simple node/edge patterns can be combined into regular expressions, by using concatenation, union or repetitions. Graph patterns, on the other hand, allow specifying the subset of matched paths that is to be returned, or joining path patterns into more complex queries. Finally, clauses/queries themselves allow us to manipulate results obtained from graph patterns, much like what is possible in the relational. Complex features such as iteration over the returned elements, passing the results to another subquery, and changing the evaluation graph, are also supported.

PATH PATTERN For $x \in \mathsf{Vars}$, $\ell \in \mathcal{L}$, $0 \leq n \leq m \in \mathbb{N}$:

$$\begin{array}{lllll}
\text{(descriptor)} & \delta & := & x \;\; \texttt{:}\ell \;\; \texttt{WHERE}\ \theta & x,\ \texttt{:}\ell,\ \text{and } \texttt{WHERE}\ \theta \text{ are optional}\\
\text{(path pattern)} & \pi & := & \texttt{(}\,\delta\,\texttt{)} & \text{(node pattern)}\\
 & & \mid & \texttt{-[}\,\delta\,\texttt{]->} \mid \texttt{<-[}\,\delta\,\texttt{]-} \mid \texttt{\~{}[}\,\delta\,\texttt{]\~{}} & \text{(edge pattern)}\\
 & & \mid & \pi\,\pi & \text{(concatenation)}\\
 & & \mid & \pi\,\texttt{|}\,\pi & \text{(union)}\\
 & & \mid & \pi\ \texttt{WHERE}\ \theta & \text{(conditioning)}\\
 & & \mid & \pi\texttt{\{}n\texttt{,}m\texttt{\}} & \text{(bounded repetition)}\\
 & & \mid & \pi\texttt{\{}n\texttt{,\}} & \text{(unbounded repetition)}
\end{array}$$

EXPRESSION and **CONDITION** For $x \in \mathsf{Vars}$, $\ell \in \mathcal{L}$, $a \in \mathcal{K}$, $c \in \mathsf{Const}$:

$$\begin{array}{llll}
\text{(expression)} & \chi & := & x \mid x\texttt{.}a \mid c\\
\text{(condition)} & \theta & := & \chi \texttt{ = } \chi \mid \chi \texttt{ < } \chi \mid \chi\ \texttt{IS NULL}\\
 & & & \mid x \texttt{ : } \ell \mid \texttt{EXISTS}\,\texttt{\{}\,\mathsf{Q}\,\texttt{\}}\\
 & & & \mid \theta\ \texttt{OR}\ \theta \mid \theta\ \texttt{AND}\ \theta \mid \texttt{NOT}\ \theta
\end{array}$$

GRAPH PATTERN For $x \in \mathsf{Vars}$:

$$\begin{array}{llll}
\text{(path mode)} & \mu & := & (\texttt{ALL} \mid \texttt{ANY})\ [\texttt{SHORTEST}]\ [\texttt{TRAIL} \mid \texttt{ACYCLIC}]\\
\text{(graph pattern)} & \Pi & := & \mu\ \ [x\ \texttt{=}]\ \ \pi \mid \Pi\texttt{,}\,\Pi
\end{array}$$

CLAUSE and **QUERY** For $k \geq 0$, $\ell \geq 1$, and $x, y, x_1, \ldots, x_k \in \mathsf{Vars}$, and $G \in \mathbb{G}$:

$$\begin{array}{llll}
\text{(clause)} & \mathsf{C} & := & \texttt{MATCH}\ \Pi\\
 & & \mid & \texttt{LET}\ x \texttt{ = } \chi\\
 & & \mid & \texttt{FOR}\ x\ \texttt{IN}\ y\\
 & & \mid & \texttt{FILTER}\ \theta\\
\text{(linear query)} & \mathsf{L} & := & \texttt{USE}\ G\ \mathsf{L}\\
 & & \mid & \mathsf{C}\ \mathsf{L}\\
 & & \mid & \texttt{RETURN}\ \chi_1\ \texttt{AS}\ x_1,\ \ldots,\ \chi_k\ \texttt{AS}\ x_k\\
\text{(query)} & \mathsf{Q} & := & \mathsf{L}\\
 & & \mid & \texttt{USE}\ G\ \texttt{\{}\mathsf{Q}_1\ \texttt{THEN}\ \mathsf{Q}_2\ \cdots\ \texttt{THEN}\ \mathsf{Q}_\ell\texttt{\}}\\
 & & \mid & \mathsf{Q}\ \texttt{INTERSECT}\ \mathsf{Q} \mid \mathsf{Q}\ \texttt{UNION}\ \mathsf{Q} \mid \mathsf{Q}\ \texttt{EXCEPT}\ \mathsf{Q}
\end{array}$$

Figure 2 Syntax of GQL.

Well-Formed Queries

The syntax of path patterns defined in Figure 2 is permissive as it allows expressions that do not type-check. For example, `(x)-[x]->()` is syntactically permitted even though it equates a node variable with an edge variable. Other patterns would provide great expressive power, such as the graph pattern `()-[y]->{0,}(), ()-[y]->{0,}*()`, which implicitly joins on lists.

We introduced in [16] a type system operating on a subset of the patterns described in Figure 2. Its goal is to ensure that GQL path patterns and graph patterns do not exhibit the pathological behavior illustrated above. Here, we will only describe the resulting syntactic restrictions informally.

Each variable is given a *type* τ from the set $\mathbb{T}$ defined by the following grammar.

$$\tau ::= \mathsf{Node} \mid \mathsf{Edge} \mid \mathsf{Path} \mid \mathsf{Maybe}(\tau) \mid \mathsf{Group}(\tau)$$

The three atomic types are used for variables returning nodes, edges, and paths, respectively. The type constructor Maybe is used for variables occurring on one side of a disjunction only, while Group is used for variables occurring under repetition, whose bindings are grouped together. As variables in pattern matching are never bound to data values, we do not need the usual types like integers or strings here.

Types are computed in a bottom-up fashion as follows. Variables appearing in node patterns (resp. in edge patterns, resp. as names of path patterns) are of type Node (resp. Edge, resp. Path). Variables appearing on one side of a disjunction with type τ but not the other are of type Maybe(τ). Variables appearing under a repetition with type τ are of type Group(τ) higher-up in the syntax tree of the expression. Consider the pattern `(-[`x`]-> | -[`y`]->){0,}`. The type of x is Edge in `-[`x`]->`, while it is Maybe(Edge) in `-[`x`]-> | -[`y`]->`, and Group(Maybe(Edge)) in `(-[`x`]-> | -[`y`]->){0,}` .

A variable x appearing in a path/graph pattern ξ is called:

- a *singleton* variable if its type is Node or Edge with respect to ξ
- a *conditional* variable if its type is Maybe(τ) for some type τ;
- a *group* variable if its type is Group(τ) for some type τ;
- a *path* variable if its type is Path.

Here is a non-exhaustive list of the syntactic conditions a pattern must meet in order for its semantics to be defined. A pattern ξ is *well-formed* if

1. Every variable appearing in a pattern ξ has one and only one type w.r.t. ξ.
2. In concatenation and join, variables appearing in both operands are singleton variables with respect to each operand.
3. In a conditioned path pattern π **WHERE** θ, every variable appearing in θ must have a type w.r.t. π.
4. In a graph pattern of the form $\mu\ \pi$ or $\mu\ x$ = π such that μ is **ALL** (which is possible since all of **SHORTEST**, **TRAIL**, and **ACYCLIC** are optional), π must contain no unbounded repetition, to avoid potentially infinite outputs.
5. For every repeated pattern π`{`n`,`m`}` or π`{`n`,}`, the *minimum path length* $\|\pi\|_{\min}$ of π, defined below, is positive. This avoids applying repetitions to paths that do not match an edge.

$$\|\nu\|_{\min} = 0 \qquad \|\pi \texttt{ WHERE } \theta\|_{\min} = \|\pi\|_{\min}$$

$$\|\eta\|_{\min} = 1 \qquad \|\pi_1 \mid \pi_2\|_{\min} = \min(\|\pi_1\|_{\min}, \|\pi_2\|_{\min})$$

$$\|\pi\texttt{\{}n,\texttt{\}}\|_{\min} = \|\pi\texttt{\{}n,m\texttt{\}}\|_{\min} = n \cdot \|\pi\|_{\min} \qquad \|\pi_1\, \pi_2\|_{\min} = \|\pi_1\|_{\min} + \|\pi_2\|_{\min}$$

Note that the local nature of types is important in item 2: implicit joins are allowed under repetitions, as in `( (a)-[]->(b)-[]->(a)-[]-> ){1,}`. Moreover, item 1 implies the existence of a schema, which is defined as follows:

▶ **Definition 1** (Schema). *A* schema *of a well-formed pattern* ξ *is a function* $\mathsf{sch}(\xi) : \mathsf{var}(\xi) \to \mathbb{T}$*, where* $\mathsf{var}(\xi)$ *is the set of variables appearing in* ξ*.*

We will assume these syntactic restrictions to be in place when defining the semantics of GQL queries in Section 4. Moreover, we define the semantics only when the computation goes as expected, that is, when it satisfies preconditions we state explicitly. For instance, we will assume that a variable is bound before being used, that we never run into clashes in variable names, and that if a specific type is expected for an operation, then the value will have that type at runtime. Some of the preconditions could be checked syntactically, at the cost of a tedious type system. Some of the preconditions cannot be checked before run-time because they depend on the data stored in the database. Deciding how to treat those cases (static analysis, runtime exceptions, implicit casts) is outside the scope of this paper. In some cases, the GQL standard describes how they should be treated, in others, they are implementation-dependent.

4 Semantics

In this section we present the formal semantics of GQL. At a high level, when evaluating a query, GQL keeps track of three things: (i) the *working graph*, which is the property graph we are using to match our patterns currently; (ii) the *working table*, that stores the information computed thus far; and (iii) the *working record*, which contains the tuple of the result we are currently using. In this section we provide mathematical abstractions for each of these concepts in order to define the semantics of GQL. We start by setting the preliminary definitions, and then move to defining the semantics for each portion of the language, as specified in Figure 2.

4.1 Preliminaries

Data model. We follow the formal definition adapted by the GQL Standard [20] to handle databases that contain multiple graphs. To define property graphs we need, in addition to the pairwise disjoint countable sets ($\mathcal{L}$ of labels, $\mathcal{K}$ of keys, and Const of constants) mentioned in Section 3, the following fresh pairwise disjoint countable sets: $\mathcal{N}$ of node ids, $\mathcal{E}_\mathsf{d}$ of directed edge ids, and $\mathcal{E}_\mathsf{u}$ of undirected edge ids.

▶ **Definition 2** (Property Graph). *A property graph is a tuple*

$$G = \langle N^G, E^G_\mathsf{d}, E^G_\mathsf{u}, \mathsf{lab}^G, \mathsf{endpoints}^G, \mathsf{src}^G, \mathsf{tgt}^G, \mathsf{prop}^G \rangle$$

where

- $N^G \subset \mathcal{N}$ *is a finite set of node ids used in* G*;*
- $E^G_\mathsf{d} \subset \mathcal{E}_\mathsf{d}$ *is a finite set of directed edge ids used in* G*;*
- $E^G_\mathsf{u} \subset \mathcal{E}_\mathsf{u}$ *is a finite set of undirected edge ids used in* G*;*
- $\mathsf{lab}^G : N^G \cup E^G_\mathsf{d} \cup E^G_\mathsf{u} \to 2^{\mathcal{L}}$ *is a labeling function that associates with every id a (possibly empty) finite set of labels from* $\mathcal{L}$*;*

- $\mathsf{src}^G, \mathsf{tgt}^G : E^G_{\mathsf{d}} \to N^G$ *define source and target of a directed edge;*
- $\mathsf{endpoints}^G : E^G_{\mathsf{u}} \to 2^N$ *so that* $|\mathsf{endpoints}^G(e)|$ *is 1 or 2 define endpoints of an undirected edge;*
- $\mathsf{prop}^G : (N^G \cup E^G_{\mathsf{d}} \cup E^G_{\mathsf{u}}) \times \mathcal{K} \to \mathsf{Const}$ *is a partial function that associates a constant with an id and a key from* $\mathcal{K}$.

If G is clear from the context, it will be omitted in the superscript. Recall that $\mathbb{G}$ denotes the set of all property graphs.

We use *node* and *edge* to refer to node ids and edge ids, respectively, and call a node u an *ℓ-node* iff $\ell \in \mathsf{lab}(u)$; similarly for edges.

▶ **Definition 3** (Graph Database). *A (property) graph database is a tuple $D = \langle G_1, \dots, G_k \rangle$ where each G_i is a property graph. We call the graph G_1 the* default graph.[2]

This is the most general definition of a database containing multiple graphs and it imposes no restrictions whatsoever on how labeling, properties, and topology agree across different graphs that share some node and edge ids. For example we may have the same id_1 for a person who has label *employee* and properties *salary, department* in a company graph and label *student* and properties *year, major* in a university graph. In fact it is even possible that the same edge id has different source and target in different graphs. We allow this complete flexibility because it is orthogonal to the choice of operations in the language, and thus we shall not impose restrictions that are not necessary for our purposes.

Paths and lists. GQL allows returning paths and lists as query answers. Here we define them formally. We start with paths.

▶ **Definition 4** (Path). *A* path *is an alternating sequence of nodes and edges that starts and ends with a node. We write paths as $p = \mathsf{path}(u_0, e_1, u_1, e_2, \cdots e_n, u_n)$, where $u_0, \dots, u_n$ are nodes, $e_1, \dots, e_n$ are (directed or undirected) edges, and $n \geq 0$. We write $\mathsf{src}(p)$ for u_0 and $\mathsf{tgt}(p)$ for u_n, and $\mathsf{len}(p)$ for its length n. We denote the set of all paths by Paths.*

For a property graph G, we say that $p \in \mathsf{Paths}$ is a path in G *if each edge in p connects the nodes before and after it in the sequence, that is, for each $i \in \{1, \dots, n\}$, at least one of the following is true:*

(a) $\mathsf{src}(e_i) = u_{i-1}$ *and* $\mathsf{tgt}(e_i) = u_i$ *in which case we speak of e_i as a* forward *edge in the path;*

(b) $\mathsf{src}(e_i) = u_i$ *and* $\mathsf{tgt}(e_i) = u_{i-1}$ *in which case we speak of e_i as a* backward *edge in the path;*

(c) $\mathsf{endpoints}(e_i) = \{u_{i-1}, u_i\}$ *in which case we speak of e_i as an* undirected *edge in the path.*

We denote the set of paths in G by $\mathsf{Paths}(G)$.

Note that we allow $n = 0$, in which case the path consists of a single vertex and no edges. Note also that in the case of a directed self-loop, both (a) and (b) in the definition above are true, hence the cases are not mutually exclusive.

▶ **Definition 5** (Concatenation of Paths). *Two paths $p = \mathsf{path}(u_0, e_0, \dots, u_k)$ and $p' = \mathsf{path}(u'_0, e'_0, \dots, u'_j)$* concatenate *if $u_k = u'_0$, in which case their* concatenation *$p \cdot p'$ is defined as $\mathsf{path}(u_0, e_0, \dots, u_k, e'_0, \dots, u'_j)$.*

[2] The default graph is used for evaluation when a specific graph is not declared by the query.

Note that a path of length 0 is a neutral element of concatenation; that is, $p \cdot \mathsf{path}(u)$ is defined iff $u = \mathsf{tgt}(p)$, in which case $p = p \cdot \mathsf{path}(u)$; likewise for $\mathsf{path}(u) \cdot p$ and $u = \mathsf{src}(p)$.

▶ **Definition 6** (List). *We use the notation* $\mathsf{list}(v_1, \ldots, v_n)$ *to denote the list containing the objects* $v_1, \ldots, v_n$ *in this order. Lists can be empty, in which case we write* $\mathsf{list}()$*. We use* Lists *to denote the set of all lists with elements in* $\mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u}$*.*

Bindings. To define the formal semantics we use bindings which specify how variables are matched to values $\mathbb{V}$ of the input graph database. Intuitively, a binding is a mathematical formalization of the concept of a working record in GQL. Formally, we set $\mathbb{V}$ as the union $\mathsf{Const} \cup \mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u} \cup \mathsf{Paths} \cup \mathsf{Lists}$.

▶ **Definition 7** (Binding). *A* binding μ *is a partial function* $\mu : \mathsf{Vars} \to \mathbb{V}$ *whose domain* $\mathrm{Dom}(\mu)$ *is finite. We denote bindings* μ *explicitly by* $(x_1 \mapsto v_1, \ldots, x_n \mapsto v_n)$ *where* $x_1, \ldots, x_n$ *are variables in* $\mathrm{Dom}(\mu)$*,* $v_1, \ldots, v_n$ *are values in* $\mathbb{V}$*, and for every* i *it holds that* $\mu(x_i) = v_i$*.*

Note that the domains of bindings are not ordered, hence for instance $(a_1 \mapsto v_1, a_2 \mapsto v_2) = (a_2 \mapsto v_2, a_1 \mapsto v_1)$. The *empty binding*, that is, the binding with an empty domain, is denoted by $()$.

▶ **Definition 8** (Compatibility of Bindings). *Two bindings* μ_1, μ_2 *are said to be compatible, denoted by* $\mu_1 \sim \mu_2$*, if they agree on their shared variables, that is, for every* $x \in \mathrm{Dom}(\mu_1) \cap \mathrm{Dom}(\mu_2)$ *it holds that* $\mu_1(x) = \mu_2(x)$*.*

If $\mu_1 \sim \mu_2$, we define their *join* $\mu_1 \bowtie \mu_2$ as expected, that is $\mathrm{Dom}(\mu_1 \bowtie \mu_2) = \mathrm{Dom}(\mu_1) \cup \mathrm{Dom}(\mu_2)$ and $(\mu_1 \bowtie \mu_2)(x) = \mu_1(x)$ whenever $x \in \mathrm{Dom}(\mu_1) \setminus \mathrm{Dom}(\mu_2)$, and $(\mu_1 \bowtie \mu_2)(x) = \mu_2(x)$ whenever $x \in \mathrm{Dom}\,(\mu_2)$.

We remark here that our definition allows joins on variables that are bound to paths or lists. However, as we will see, the syntactic restrictions on queries limit this feature significantly.

4.2 Semantics of Path Patterns

We start by defining the semantics of path patterns. For the remainder of this subsection, we consider a fixed property graph

$$G = \langle N^G, E^G_\mathsf{d}, E^G_\mathsf{u}, \mathsf{lab}^G, \mathsf{endpoints}^G, \mathsf{src}^G, \mathsf{tgt}^G, \mathsf{prop}^G \rangle.$$

Moreover, we assume that all queries are well-formed and all patterns considered are restricted syntactically as described in Section 3. The semantics $[\![\pi]\!]_G$ of a pattern π is a set of pairs (p, μ) where μ a binding, and p is a path in G. In $[\![\pi]\!]_G$, G denotes the working graph in GQL parlance (specified by the keyword **USE**), and the pairs (p, μ) model what is computed over this working graph.

Semantics of Node and Edge Patterns

$$[\![\texttt{()}]\!]_G \;= \left\{\, (n, ()) \;\middle|\; n \in N^G \,\right\} \qquad\qquad [\![\texttt{(}x\texttt{)}]\!]_G = \left\{\, (n, (x \mapsto n)) \;\middle|\; n \in N^G \,\right\}$$

$$[\![\texttt{(:}\ell\texttt{)}]\!]_G = \left\{\, (n, ()) \;\middle|\; n \in N^G,\ \ell \in \mathsf{lab}^G(n) \,\right\}$$

Other cases are treated by moving the label and conditions outside of the node pattern. For instance, `(`x`:`ℓ **WHERE** θ`)` is rewritten as `(`x`)` **WHERE** `(`x`:`ℓ **AND** θ`)`.

$$
\begin{aligned}
[\![\texttt{-[]->}]\!]_G &= \left\{ (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), ()) \;\middle|\; e \in E_d^G \right\} \\
[\![\texttt{-[}x\texttt{]->}]\!]_G &= \left\{ (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), (x \mapsto e)) \;\middle|\; e \in E_d^G \right\} \\
[\![\texttt{-[:}\ell\texttt{]->}]\!]_G &= \left\{ (\mathsf{path}(\mathsf{src}(e), e, \mathsf{tgt}(e)), ()) \;\middle|\; e \in E_d^G, \ell \in \mathsf{lab}^G(e) \right\}
\end{aligned}
$$

Other cases of the forward edge patterns are treated by moving the label and conditions outside of the edge pattern, just as for node patterns. Backward edge patterns and undirected edge patterns are treated similarly, with the base cases given below.

$$
\begin{aligned}
[\![\texttt{<-[]-}]\!]_G &= \left\{ (\mathsf{path}(\mathsf{tgt}(e), e, \mathsf{src}(e)), ()) \;\middle|\; e \in E_d^G \right\} \\
[\![\texttt{~[]~}]\!]_G &= \left\{ (\mathsf{path}(u_1, e, u_2), ()), (\mathsf{path}(u_2, e, u_1), ()) \;\middle|\; \begin{array}{l} e \in E_u^G \\ \{u_1, u_2\} = \mathsf{endpoints}^G(e) \end{array} \right\}
\end{aligned}
$$

Semantics of Concatenation, Union, and Conditioning

$$
[\![\pi_1\, \pi_2]\!]_G \left\{ (p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \;\middle|\; \begin{array}{l} (p_i, \mu_i) \in [\![\pi_i]\!]_G \text{ for } i = 1, 2 \\ p_1 \text{ and } p_2 \text{ concatenate} \\ \mu_1 \sim \mu_2 \end{array} \right\}
$$

Note that since $\pi_1\, \pi_2$ is assumed to be well-formed, all variables shared by π_1 and π_2 are singleton variables (Condition 2 in Section 3). In other words, implicit joins over group and optional variables are disallowed; the same remark will also apply for the semantics of joins.

▶ Remark 9. Consider the pattern

```
(x) (-[:Transfer]->()-[:Transfer]->(x)]){1,}
```

This pattern is disallowed in GQL because the leftmost `x` is a singleton variable, whereas the rightmost `x` is a group variable. In GQL philosophy, the leftmost `x` will be bound to a node and the rightmost `x` will be bound to a list of nodes, which is a type mismatch.

$$
[\![\pi_1 \mid \pi_2]\!]_G = \{ (p, \mu \cup \mu') \mid (p, \mu) \in [\![\pi_1]\!]_G \cup [\![\pi_2]\!]_G \}
$$

where μ' maps every variable in $\mathsf{var}(\pi_1 \mid \pi_2) \setminus \mathrm{Dom}(\mu)$ to null. (Recall that var maps a pattern to the set of variables appearing in it.)

$$
[\![\pi \ \texttt{WHERE}\ \theta]\!]_G = \{ (p, \mu) \in [\![\pi]\!]_G \mid [\![\theta]\!]_G^\mu = \mathsf{true} \}
$$

Semantics of Repetition

$$
\begin{aligned}
[\![\pi\texttt{\{}n, m\texttt{\}}]\!]_G &= \bigcup_{i=n}^{m} [\![\pi]\!]_G^i \\
[\![\pi\texttt{\{}n,\texttt{\}}]\!]_G &= \bigcup_{i=n}^{\infty} [\![\pi]\!]_G^i
\end{aligned}
$$

Above, for a pattern π and a natural number $i \geq 0$, we use $[\![\pi]\!]_G^i$ to denote the i-th power of $[\![\pi]\!]_G$, which we define as

$$
[\![\pi]\!]_G^0 = \{ (\mathsf{path}(u), \mu) \mid u \text{ is a node in } G \}
$$

where μ binds each variable in $\mathrm{Dom}(\mathsf{sch}(\pi))$ to $\mathsf{list}()$, that is, the empty-list value; and

$$
\forall i > 0 \quad [\![\pi]\!]_G^i = \left\{ (p_1 \cdot \ldots \cdot p_i, \mu') \;\middle|\; \begin{array}{l} (p_1, \mu_1), \ldots, (p_n, \mu_i) \in [\![\pi]\!]_G \\ p_1, \ldots, p_i \text{ concatenate} \end{array} \right\}
$$

where μ' binds each variable in $\mathrm{Dom}(\mathsf{sch}(\pi))$ to $\mathsf{list}\big(\mu_1(x), \ldots, \mu_i(x)\big)$. Recall that sch is defined in Section 3.

▶ Remark 10. Since $\pi\texttt{\{}n\texttt{,\}}$ is assumed to be well-formed, it holds $\|\pi\|_{\min} \geq 1$. A simple induction then yields that each p_i in the definition above has positive length. A second induction then yields that, given a path p, there are finitely many assignments μ such that $(p,\mu) \in [\![\pi\texttt{\{}n\texttt{,}m\texttt{\}}]\!]_G$. This fact is crucial to have a finite output in the end.

For instance, consider a graph with a single node u and no edges, and the pattern `(a){0,}` which is not well-formed (the minimal path length of `()` is 0). For every i, the set $[\![\texttt{(a)}]\!]^i_G$ contains $(\mathsf{path}(u), \mu_i)$ where $\mu_i = (a \mapsto \mathsf{list}(\underbrace{u,\dots,u}_{i \text{ times}}))$; hence the union in the definition of $[\![\pi\texttt{\{}n\texttt{,\}}]\!]_G$ above would not only yield an infinite number of elements, but all of them would be associated to the same path. As a result a graph pattern such as `ALL SHORTEST (a){0,}` would have infinitely many results.

4.3 Semantics of Graph Patterns

We now define the semantics of graph patterns. We first fully define atomic graph patterns and then define their joins.

$$[\![x\texttt{=}\ \pi]\!]_G = \big\{(p, \mu \cup \{x \mapsto p\}) \mid (p,\mu) \in [\![\pi]\!]_G\,\big\}$$

In the following we denote by $\tilde{\pi}$ a graph pattern that never uses the “`,`” operator, hence it is of the form $\mu\ x\texttt{=}\ \pi$, where μ is a path mode, x is a variable, π is a path pattern, and “x=“ is optional.

$$\begin{aligned}
[\![\texttt{TRAIL}\ \pi]\!]_G &= \{\,(p,\mu) \in [\![\pi]\!]_G \mid \text{no edge occurs more than once in } p\,\}\\
[\![\texttt{ACYCLIC}\ \pi]\!]_G &= \{\,(p,\mu) \in [\![\pi]\!]_G \mid \text{no node occurs more than once in } p\,\}\\
[\![\texttt{SHORTEST}\ \tilde\pi]\!]_G &= \left\{ (p,\mu) \in [\![\tilde\pi]\!]_G \;\middle|\; \mathsf{len}(p) = \min\left\{ \mathsf{len}(p') \;\middle|\; \begin{array}{l} (p',\mu') \in [\![\tilde\pi]\!]_G\\ \mathsf{src}(p') = \mathsf{src}(p)\\ \mathsf{tgt}(p') = \mathsf{tgt}(p) \end{array} \right\} \right\}\\
[\![\texttt{ALL}\ \tilde\pi]\!]_G &= [\![\tilde\pi]\!]_G\\
[\![\texttt{ANY}\ \tilde\pi]\!]_G &= \bigcup_{(s,t)\in X} \{\mathsf{any}(\{\,(p,\mu) \mid (p,\mu) \in [\![\tilde\pi]\!]_G\,, \mathsf{endpoints}(p) = (s,t)\,\}\}
\end{aligned}$$

where $X = \{\,\big(\mathsf{src}(p), \mathsf{tgt}(p)\big) \mid (p,\mu) \in [\![\tilde\pi]\!]_G\,\}$ and any is a procedure that arbitrarily returns one element from a set; any need not be deterministic.

$$[\![\Pi_1\texttt{,}\ \Pi_2]\!]_G = \{\,(\bar p_1 \times \bar p_2, \mu_1 \bowtie \mu_2) \mid (\bar p_i, \mu_i) \in [\![\Pi_i]\!]_G \ \text{ for } i = 1,2 \text{ and } \mu_1 \sim \mu_2\,\}$$

Here, $\bar p_1 = (p_1^1, p_1^2, \dots, p_1^k)$ and $\bar p_2 = (p_2^1, p_2^2, \dots, p_2^l)$ are tuples of paths, and $\bar p_1 \times \bar p_2$ stands for $(p_1^1, p_1^2, \dots, p_1^k, p_2^1, p_2^2, \dots, p_2^l)$. Just as it is the case of concatenation, since $\Pi_1\texttt{,}\ \Pi_2$ is well-formed, implicit joins can occur over singleton variables only.

4.4 Semantics of Conditions and Expressions

The semantics $[\![\chi]\!]^\mu_G$ of an expression χ is an element in $\mathbb{V}$ that is computed with respect to a binding μ and a graph G. Intuitively, variables in χ are evaluated with μ and we use G to access the properties of an element. It is formally defined as follows.

$$\begin{aligned}
[\![c]\!]^\mu_G &= c \qquad \text{for } c \in \mathsf{Const}\\
[\![x]\!]^\mu_G &= \mu(x) \qquad \text{for } x \in \mathrm{Dom}(\mu)\\
[\![x.a]\!]^\mu_G &= \begin{cases} \mathsf{prop}^G(\mu(x), a) & \text{if } (\mu(x), a) \in \mathrm{Dom}(\mathsf{prop}^G)\\ \mathsf{null} & \text{else if } \mu(x) \in (\mathcal{N} \cup \mathcal{E}_{\mathsf{d}} \cup \mathcal{E}_{\mathsf{u}}) \end{cases} \qquad \text{for } x \in \mathrm{Dom}(\mu), a \in \mathcal{K}
\end{aligned}$$

▶ Remark 11. Recall that different graphs may share nodes and edges. Hence the condition $(\mu(x), a) \in \mathrm{Dom}(\mathsf{prop}^G)$, above, does imply that $\mu(x)$ is a node or an edge in G, but does **not** imply that it was matched in G.

The semantics $\llbracket\theta\rrbracket_G^\mu$ of a condition θ is an element in $\{\mathsf{true}, \mathsf{false}, \mathsf{null}\}$ that is evaluated with respect to a binding μ and a graph G, and is defined as follows:

$$\llbracket\chi_1 \texttt{ = } \chi_2\rrbracket_G^\mu = \begin{cases} \mathsf{null} & \text{if } \llbracket\chi_1\rrbracket_G^\mu = \mathsf{null} \text{ or } \llbracket\chi_2\rrbracket_G^\mu = \mathsf{null} \\ \mathsf{true} & \text{if } \llbracket\chi_1\rrbracket_G^\mu = \llbracket\chi_2\rrbracket_G^\mu \neq \mathsf{null} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\llbracket\chi_1 \texttt{ < } \chi_2\rrbracket_G^\mu = \begin{cases} \mathsf{null} & \text{if } \llbracket\chi_1\rrbracket_G^\mu = \mathsf{null} \text{ or } \llbracket\chi_2\rrbracket_G^\mu = \mathsf{null} \\ \mathsf{true} & \text{else if } \llbracket\chi_1\rrbracket_G^\mu < \llbracket\chi_2\rrbracket_G^\mu \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\llbracket\chi \texttt{ IS NULL}\rrbracket_G^\mu = \begin{cases} \mathsf{true} & \text{if } \llbracket\chi\rrbracket_G^\mu = \mathsf{null} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\llbracket\chi\texttt{:}\ell\rrbracket_G^\mu = \begin{cases} \mathsf{true} & \text{if } \llbracket\chi\rrbracket_G^\mu \in N^G \cup E_u^G \cup E_d^G \text{ and } \ell \in \mathsf{lab}^G(\llbracket\chi\rrbracket_G^\mu) \\ \mathsf{false} & \text{else if} \llbracket\chi\rrbracket_G^\mu \in \mathcal{N} \cup \mathcal{E}_\mathsf{d} \cup \mathcal{E}_\mathsf{u} \end{cases}$$

$$\llbracket\theta_1 \texttt{ AND } \theta_2\rrbracket_G^\mu = \llbracket\theta_1\rrbracket_G^\mu \wedge \llbracket\theta_2\rrbracket_G^\mu \quad ^{(*)}$$

$$\llbracket\theta_1 \texttt{ OR } \theta_2\rrbracket_G^\mu = \llbracket\theta_1\rrbracket_G^\mu \vee \llbracket\theta_2\rrbracket_G^\mu \quad ^{(*)}$$

$$\llbracket\texttt{NOT } \theta\rrbracket_G^\mu = \neg\, \llbracket\theta\rrbracket_G^\mu \quad ^{(*)}$$

$^{(*)}$ Operators $\wedge$, $\vee$, and $\neg$ are defined as in SQL three-valued logic, e.g. $\mathsf{null} \vee \mathsf{true} = \mathsf{true}$ while $\mathsf{null} \wedge \mathsf{true} = \mathsf{null}$.

$$\llbracket\texttt{EXISTS \{}\,\mathsf{Q}\,\texttt{\}}\rrbracket_G^\mu = \begin{cases} \mathsf{true} & \text{if } \llbracket\mathsf{Q}\rrbracket_G(\{\mu\}) \text{ is not empty} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

4.5 Semantics of Queries

Clauses and queries are interpreted as functions that operate on tables. These tables are our abstraction of GQL's working tables.

▶ **Definition 12.** *A* table T *is a set of bindings that have the same domains, referred to as* $\mathrm{Dom}(T)$.

Note that tables do not have schemas: two different bindings in a table might associate a variable to values of incompatible types.

Semantics of Clauses

The semantics $\llbracket\mathsf{C}\rrbracket_G$ of a clause C is a function that maps tables into tables, and is parametrized by a graph G. Patterns, conditions and expression in a clause are evaluated with respect to that G.

$$\llbracket\texttt{MATCH } \Pi\rrbracket_G(T) = \bigcup_{\mu\in T}\{\mu \bowtie \mu' \mid (p, \mu') \in \llbracket\Pi\rrbracket_G,\ \mu \sim \mu'\}$$

Note that if Π uses a variable that already occurs in $\mathrm{Dom}(T)$, a join is performed. Unlike in the case of path patterns and graph patterns, this join can involve variables bound to lists or paths. While this is not problematic mathematically, it might be disallowed in future iterations of GQL.

If $x \notin \mathrm{Dom}(T)$, then

$$\llbracket \texttt{LET}\ x\ \texttt{=}\ \chi \rrbracket_G (T) = \bigcup_{\mu \in T} \{\mu \bowtie (x \mapsto \llbracket \chi \rrbracket_G^{\mu})\}$$

$$\llbracket \texttt{FILTER}\ \theta \rrbracket_G (T) = \bigcup_{\mu \in T} \{\mu \mid \llbracket \theta \rrbracket_G^{\mu} = \mathsf{true}\} \ .$$

If $x \notin \mathrm{Dom}(T)$ and, for every $\mu \in T$, $\mu(y)$ is a list or null,[3] then

$$\llbracket \texttt{FOR}\ x\ \texttt{IN}\ y \rrbracket_G (T) = \bigcup_{\mu \in T} \{\mu \bowtie (x \mapsto v) \mid v \in \mu(y)\} \ .$$

Semantics of Linear Queries

$$\begin{aligned} \llbracket \texttt{USE}\ G'\ \mathsf{L} \rrbracket_G (T) &= \llbracket \mathsf{L} \rrbracket_{G'} (T) \\ \llbracket C\ \mathsf{L} \rrbracket_G (T) &= \llbracket \mathsf{L} \rrbracket_G \big(\llbracket C \rrbracket_G (T) \big) \\ \llbracket \texttt{RETURN}\ \chi_1\ \texttt{AS}\ x_1, \ldots, \chi_\ell\ \texttt{AS}\ x_\ell \rrbracket_G (T) &= \bigcup_{\mu \in T} \{(x_1 \mapsto \llbracket \chi_1 \rrbracket_G^{\mu}, \ldots, x_\ell \mapsto \llbracket \chi_\ell \rrbracket_G^{\mu})\} \end{aligned}$$

Semantics of Queries

The *output of a query* Q is defined as

$$\mathsf{Output}(\mathsf{Q}) = \llbracket \mathsf{Q} \rrbracket_G (\{()\}) \ ,$$

where $\{()\}$ is the unit table that consists of the empty binding, and G is the default graph in D. We define the semantics of queries recursively as follows.

$$\llbracket \texttt{USE}\ G'\ \texttt{\{}\mathsf{Q}_1\ \texttt{THEN}\ \mathsf{Q}_2\ \cdots\ \texttt{THEN}\ \mathsf{Q}_k\texttt{\}} \rrbracket_G (T) = \llbracket \mathsf{Q}_k \rrbracket_{G'} \circ \cdots \circ \llbracket \mathsf{Q}_1 \rrbracket_{G'} (T)$$

If $\mathrm{Dom}\left(\llbracket \mathsf{Q}_1 \rrbracket_G (T)\right) = \mathrm{Dom}\left(\llbracket \mathsf{Q}_2 \rrbracket_G (T)\right)$, then we let

$$\begin{aligned} \llbracket \mathsf{Q}_1\ \texttt{INTERSECT}\ \mathsf{Q}_2 \rrbracket_G (T) &= \llbracket \mathsf{Q}_1 \rrbracket_G (T) \cap \llbracket \mathsf{Q}_2 \rrbracket_G (T) \\ \llbracket \mathsf{Q}_1\ \texttt{UNION}\ \mathsf{Q}_2 \rrbracket_G (T) &= \llbracket \mathsf{Q}_1 \rrbracket_G (T) \cup \llbracket \mathsf{Q}_2 \rrbracket_G (T) \\ \llbracket \mathsf{Q}_1\ \texttt{EXCEPT}\ \mathsf{Q}_2 \rrbracket_G (T) &= \llbracket \mathsf{Q}_1 \rrbracket_G (T) \setminus \llbracket \mathsf{Q}_2 \rrbracket_G (T) \end{aligned}$$

5 A Few Known Discrepancies with the GQL Standard

In pursuing the goal of introducing the key features of GQL to the research community, we inevitably had to make decisions that resulted in discrepancies between our presentation and the 500+ pages of the forthcoming Standard. In this section, we discuss a non-exhaustive list of differences between the actual GQL Standard and our digest. To start with, in all our formal development we assumed that queries are given by their syntax trees, which result from parsing them. Hence we completely omitted such parsing-related aspects as parentheses, operator precedence etc. Also we note that many GQL features, even those described here, are optional, and not every implementation is obliged to have them all.

[3] Note that null is treated just as list()

The remaining discrepancies are divided into three main categories: syntactic restrictions (Section 5.1), query evaluation (Section 5.2), and missing features (Section 5.3). The reader must bear in mind that, as the GQL Standard is roughly one year from publication in its final form, many aspects of the language may still change in a way that depends on the work of the Committee, and thus is impossible to predict.

5.1 User-Friendly Syntactic Restrictions

The GQL Standard imposes restrictions on the syntax that aim at preventing unexpected behavior, and that we generally did not describe. Two such examples are given below.

First, consider the queries $Q_1 =$ `MATCH` μ `x=-[]->*` and $Q_2 =$ `MATCH` μ `x=-[]->*()` for some path mode μ (it does not matter which one). According to our semantics, both return one binding, namely $(x \mapsto \texttt{path}(u))$, for each node u in the graph; however, Q_1 is syntactically forbidden in the GQL Standard because no node pattern occurs. Another interesting syntactic restriction concerns *strict interior* variables under selectors, such as `c` in the following:

```
MATCH ANY (:Person) -[]->* (c:Account) -[]->* (:Person),
      ANY (:Person) -[]->* (c:Account) -[]->* (:Person)
```

The `ANY` selectors are evaluated independently, and before the implicit join on variable `c`. Then, the node bound to the variable `c` by either path pattern is arbitrary, and joining on them is very likely to fail. This situation was not deemed user-friendly by the Committee, and therefore precluded.

5.2 Query Evaluation

Bag semantics. For simplicity, we described GQL as if it was following set semantics but, in reality, GQL uses bags just like Cypher and SQL. In order to define clauses and queries under bag semantics, small changes are needed:

- tables should be defined as bags, rather than sets, of bindings;
- unions ($\cup$) over the elements of a table should be additive bag unions ($\uplus$); and
- set comprehensions should be replaced with bag comprehensions.

As an example, if we denote bags with double curly braces, then the semantics of `RETURN` is

$$[\![\texttt{RETURN}\ \chi_1\ \texttt{AS}\ x_1, \ldots, \chi_\ell\ \texttt{AS}\ x_\ell]\!]_G\,(T) = \biguplus_{\mu \in T} \{\!\{(x_1 \mapsto [\![\chi_1]\!]^{\mu}_{G}, \ldots, x_\ell \mapsto [\![\chi_\ell]\!]^{\mu}_{G})\}\!\}$$

Note that GQL partially eliminates duplicates during pattern matching, which is reflected here by the semantics of graph patterns: $[\![\Pi]\!]_G$ is a *set* of path/binding pairs, while $[\![\texttt{MATCH}\ \Pi]\!]_G$ returns a *bag* of bindings by projecting out the paths (see the definition of $[\![\texttt{MATCH}\ \Pi]\!]_G$ in Section 4.5). Hence, different ways to compute the same path/binding pair will only contribute to one copy of the binding in the output of $[\![\texttt{MATCH}\ \Pi]\!]_G$. It is still possible to get multiple copies of some binding in the output, but these come from pairs with different paths.

Partial deduplication is an effort to unify the multiplicies of queries that express *the same* pattern in different ways. To see this, consider the queries

Q_1: `MATCH (a:Person)-[]->(b WHERE b:Person OR b:Account)`

Q_2: `MATCH (a:Person)-[]->(b:Person) | (a)-[]->(b:Account)`

and the path (v_1, e_1, v_2) matched by either of them with the binding $\mu_1 = (a \to v_1, b \mapsto v_2)$, where v_2 bears *both* labels `Person` and `Account`. As the disjunction in Q_1 is expressed using a Boolean condition, this query always returns a single copy of μ_1. In Q_2, however, the disjunction is expressed with a union (`|`) of patterns; thus, if the semantics of `|` were defined as a bag-union, the query would return two copies of μ_1.

Finally, as in SQL, the operations `INTERSECT`, `UNION`, and `EXCEPT` remove duplicates in GQL, while the variants `INTERSECT ALL`, `UNION ALL`, and `EXCEPT ALL` do not.

Path bindings. In a nutshell, a *path binding* is a path where each element may be annotated with variables, and it is inconsistent as soon as two different elements have the same annotation (see [12] for details). Thus, a path binding defines a single path/binding pair, whereas a path/binding pair can define several path bindings. In GQL Standard, pattern matching computes a set of consistent path bindings, while our semantics computes a set of path/binding pairs, and the results are bags formed by projecting away paths. Consequently, our semantics might sometimes return fewer results than GQL's, but the difference only affects multiplicity. For example, consider `MATCH ()-[]->(a) | (a)-[]->()` on a graph with a single node u and a single (looping) edge. According to our semantics, only one copy of $(a \mapsto u)$ is returned, while two occurrences of it are returned according to GQL Standard.

Postponed evaluation of conditions. In our treatment of the language, the semantics of the following query is undefined:

```
MATCH ALL SHORTEST -[x]-> ( ()-[y]->() WHERE x.amount < y.amount ){10,10}
```

Indeed, when the condition `WHERE x.amount < y.amount` is evaluated, the variable `x` is not yet bound, as `-[x]->` occurs in a different branch of the query's syntax tree. In GQL Standard, however, the above query is legal, because the evaluation of `WHERE` conditions is postponed for as long as possible.[4] While the meaning of the query is clear, its evaluation is non-trivial. The context of each condition (here, `y` is bound to ten successive edge ids) must be recorded, because it will be different when the evaluation occurs. Note that the evaluation of conditions must occur before the evaluation of `SHORTEST`, hence queries like

```
MATCH -[x]->, ALL SHORTEST ( -[y]-> WHERE x.amount < y.amount ){10,10}
```

are not allowed in GQL.

Referencing the input table in conditions during pattern patching. In our semantics, the input table is not passed on to pattern matching, so one cannot refer to variables from it in `WHERE` conditions. As an example, the semantics of `LET x=42 MATCH (a WHERE a.amount=x)` is undefined. It is not yet clear whether such a query is allowed in the GQL Standard or not.

5.3 Missing Features

Syntactic sugar. The GQL Standard includes a lot of syntactic sugar that we disregarded. For instance, several other types of edge patterns exist, such as `-[`δ`]-`, which matches edges regardless of their direction. Another example is the possibility of using `*` and `+` as shorthands for `{0,}` and `{1,}`, respectively.

Complex label expressions. We only allow a single label in descriptors, but the GQL Standard allows complex label expressions, as in `MATCH (a:YachtClub|(Person&!Account))`. Using `WHERE`, this could be rewritten as

```
MATCH (a WHERE a:YachtClub OR (a:Person AND NOT a:Account))
```

Label expressions can also use the special atom "`%`" to check the nonemptyness of the label set. For example, `MATCH (a:%)` matches nodes with at least one label and `MATCH (a:!%)` matches node with no labels. Note that "`%`" cannot be used to define a regular expression of labels, unlike its usage in the `LIKE` expressions of SQL.

[4] This is orthogonal to left-to-right evaluation: `-[x]->` could be placed on the right instead.

Complex path modes. GQL allows more complex path modes than described here. Recall that `SHORTEST` partitions matched paths by endpoints and returns the shortest paths for each pair of endpoints. `SHORTEST` k `GROUPS` generalizes this: for each pair of endpoints, it returns all paths of length at most i_k, where $i_1 < i_2 < \cdots < i_k$ are the k smallest lengths of paths between these endpoints. `SHORTEST k PATHS` returns k shortest paths for each pair of endpoints. Another mode present in GQL is `SIMPLE`: it is similar to `ACYCLIC` but allows the first and the last node on a path to be the same, i.e., a simple cycle. There is also the keyword `WALK` to explicitly indicate the absence of a path mode.

GQL's `TRAIL` differs from Cypher's *trail semantics* [18, 17]. The latter corresponds to GQL's *match mode* `DIFFERENT EDGES`, which is omitted in this digest. Indeed, Cypher's requirement that all matched edges must be different operates at the level of graph patterns, whereas GQL's `TRAIL` operates at the level of path patterns. Hence, while the GQL query `MATCH TRAIL ()-[e1]->(), TRAIL ()-[e2]->()` will return bindings in which `e1` and `e2` are equal, the Cypher query `MATCH ()-[e1]>(), ()-[e2]->()` would not; the latter behaviour is captured by the GQL query `MATCH DIFFERENT EDGES ()-[e1]>(), ()-[e2]->()`.

Finally, we only use path modes at the beginning of path patterns. GQL's rules are more involved, in that they allow `TRAIL` and `ACYCLIC` to be used inside patterns.

Projection clauses. The GQL Standard includes several clauses similar to `RETURN`, such as `YIELD`, `PROJECT`, and `SELECT`. We ignored these because, although they are not allowed at the same positions in queries, they can be simulated by simple rewritings in terms of `RETURN`.

Combination of queries. In addition to set operations (`UNION`, etc.) and bag operations (`UNION ALL`, etc.), queries could be of the form Q_1 `OTHERWISE` Q_2. Its semantics is as follows: $[\![\mathsf{Q}_1$ `OTHERWISE` $\mathsf{Q}_2]\!]\,(T)$ equals $[\![\mathsf{Q}_1]\!]\,(T)$ if table $[\![\mathsf{Q}_1]\!]\,(T)$ is non-empty, otherwise it equals $[\![\mathsf{Q}_2]\!]\,(T)$.

Aggregation. The GQL Standard will feature two kinds of aggregation. The first one, much like `GROUP BY` in SQL, groups together bindings under which the evaluation of an expression produces the same value, then an aggregate value is computed for each group. The exact details are still under development, but it appears likely that such aggregation will be limited to `RETURN` statements, thus having a very relational character.

The second kind will aggregate along matched paths to compute a value, both during and after pattern matching. Computing the length of a path is a typical example; one can have more complex aggregates, such as the sum of the values `n.amount` for each node `n` in the path. This is similar to `reduce` in Cypher. The use of this feature in pattern matching requires strong syntactic restrictions for query evaluation to be decidable [16].

Subqueries. GQL has a facility to run subqueries through the `CALL` Q clause, the semantics of which is roughly as follows: for each binding μ in the input table, $[\![Q]\!]_G\,(\{\mu\})$ is evaluated in a sub-process, and the resulting table is left-joined with the current working table. An important detail is that `CALL` can only expand bindings. It cannot remove columns from the input table nor change the values in them. The existence of read-only columns matters in clauses like `RETURN`, which cannot therefore be treated with our semantics as is. In GQL, this is handled with a notion of *working record.*

Note also that `CALL` Q will make nondeterminism much harder to detect if updates happen in Q. Tables are unordered sets (or bags) but in an update clause each binding causes changes in the graph (see next item) and so it can modify the evaluation of the clause for the next binding. In such cases, inconsistent changes may be detected [21].

Updates. Graph database updates in GQL are outside the scope of this paper. They will work similarly to Cypher updates [21], by using clauses that can add and remove elements (`INSERT` and `DELETE`), or modify elements' attributes (`SET` and `REMOVE`). Therefore, pattern matching and updates can be mixed together and result in bulk updates to the graph based on its contents, as in the example below:

```
MATCH  (a:Account)         -- match every Account a
INSERT (p:Person)          -- create a new Person node for each a
SET     p.name = a.owner   -- set the name of the new node
INSERT (p)-[:Owns]->(a)    -- create a new "Owns" edge from p to a
REMOVE  a.owner            -- remove the owner property from a
```

6 What the Future Holds

In this paper we have summarized the key elements of GQL, which is currently being developed as a new standard graph query language (the timeline of ISO calls for the publication of the Standard in either late 2023 or early 2024). At the time when the first version of the SQL Standard was produced, many key elements of relational theory were already in place. For GQL, the standardization work is well ahead of the academic developments it should ideally be based upon. In what follows, we bring to the attention of the community several directions of academic work that will facilitate the development of graph query languages and their standardization.

Expressiveness and complexity. For relational query languages, the database research community has uncovered a rich landscape of fragments (conjunctive queries, positive queries, and queries with inequalities are some very well studied examples) and extensions (for example, adding counting and aggregation, or adding recursion as in many instantiations of datalog), see [1, 4]. For these, we understand the trade-off between their expressiveness and the complexity of query evaluation. Here we have described a basic language for graphs, essentially the core of GQL, akin to relational algebra and calculus. Now we need to develop its theory, starting with understanding expressiveness and complexity and their trade-offs, in a way similar to what we know about relational databases. For the pattern matching facilities of GQL, shared with SQL/PGQ, some early results are available [16].

Query processing and optimization. Query processing and optimization is a central area in relational database research that needs yet to be developed for GQL. In a more theoretical level, the basis for understanding optimization is query equivalence and containment. We know a thing or two about containment for (conjunctive) regular path queries [9, 15] and extensions with data [26] but not for queries that resemble the real-life language. Moving to more practical aspects, one needs efficient and practical algorithms and data structures for processing graph queries in GQL, whether in a native system, or a relational implementation. Of course there is significant work in this direction [37, 23, 5, 28, 35, 31, 25] but it needs to be adjusted to languages that will dominate the practical landscape for decades.

Design decisions and alternatives. We explained in Remark 9 how GQL currently forbids concatenating patterns that contain different kinds of variables. Notice, however, that this current state reflects a design decision and it may be interesting to explore other avenues for graph query languages. For instance, one could consider a semantics in which both occurrences of `x` in Remark 9 should be bound to single nodes. Under such a semantics, the pattern would essentially perform a join on the even nodes of the path and would match "flower" shaped paths centered around node `x`, consisting of Transfer-loops of length two. Alternatively, one could consider a semantics in which, as soon as `x` occurs as a group variable, all occurrences of `x` are considered to be group variable occurrences. In this case, the query would match Transfer-paths of even length and bind `x` to the list of "even" nodes on such paths. In line with this work would be the study of an automaton model with group variables that would allow classical evaluation and automata-theoretic constructions such as the product, determinization, etc. Since GQL is a complex language, there are many such places in which fundamental research can either help to validate the current design decisions or propose alternatives.

Updates. We have concentrated on the read-only part of the languages and have not touched updates. Designing a proper update language is not a simple task: in Cypher, for example, the initial design exhibited a multitude of problems [21]. GQL largely follows Cypher, which means its updates and transaction processing facilities need to be designed with care and subjected to the same research scrutiny as their relational counterpart.

Graph-to-graph queries. GQL, as its precursors including Cypher, is a very good tool for turning graphs into relations. The ever reappearing issue in the field of graph languages is how to design a graph-to-graph language whose queries output graphs. Queries are then composable: a query can be applied to the output of a previous one. We also regain such basic concepts as views and subqueries, taken for granted in relational databases, but very limited in the current graph database landscape.

Metadata. Looking into the future, we need to have a good schema language for graphs, and see how it interacts with graph query languages. Some efforts in this direction have already been made: for example, the PG-Keys proposal introduces keys for property graphs [3] and more recently proposed PG-Schema [2] specifies a schema language for property graphs that should lead to future schema standards. As these are formulated, much theory needs to be developed, for example semantic query optimization, as well as incremental validation of schemas and constraints following work for relational and semistructured data [24, 6].

References

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

2 Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. PG-Schema: Schemas for property graphs, 2022. `arXiv:2211.10962`.

3 Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. PG-Keys: Keys for property graphs. In *SIGMOD '21: International Conference on Management of Data*, pages 2423–2436. ACM, 2021. `doi:10.1145/3448016.3457561`.

4 Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. *Database Theory.* Open source at `https://github.com/pdm-book/community`, 2022.

5 Jorge A. Baier, Dietrich Daroch, Juan L. Reutter, and Domagoj Vrgoč. Evaluating navigational RDF queries over the web. In *HT*, pages 165–174. ACM, 2017. `doi:10.1145/3078714.3078731`.

6 Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient incremental validation of XML documents. In *ICDE*, pages 671–682. IEEE Computer Society, 2004. doi:10.1109/ICDE.2004.1320036.

7 Véronique Benzaken and Evelyne Contejean. A coq mechanised formal semantics for realistic SQL queries: formally reconciling SQL and bag relational algebra. In *CPP*, pages 249–261. ACM, 2019. doi:10.1145/3293880.3294107.

8 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR*, pages 176–185. Morgan Kaufmann, 2000.

9 Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Reasoning on regular path queries. *SIGMOD Rec.*, 32(4):83–92, 2003. doi:10.1145/959060.959076.

10 Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017. doi:10.1145/3062341.3062348.

11 Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *SIGMOD Conference*, pages 323–330. ACM Press, 1987. doi:10.1145/38713.38749.

12 Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD Conference*, pages 2246–2258. ACM, 2022. doi:10.1145/3514221.3526057.

13 Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. Aggregation support for modern graph analytics in TigerGraph. In *SIGMOD Conference*, pages 377–392. ACM, 2020. doi:10.1145/3318464.3386144.

14 Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for a web-site management system. *SIGMOD Rec.*, 26(3):4–11, 1997. doi:10.1145/262762.262763.

15 Diego Figueira, Adwait Godbole, Shankara Narayanan Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. Containment of simple conjunctive regular path queries. In *KR*, pages 371–380, 2020. doi:10.24963/kr.2020/38.

16 Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. GPC: A pattern calculus for property graphs. In *PODS'23*, 2023. To appear.

17 Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. Formal semantics of the language Cypher, 2018. arXiv:1802.09984.

18 Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD Conference*, pages 1433–1445. ACM, 2018. doi:10.1145/3183713.3190657.

19 GQL influence graph. https://www.gqlstandards.org/existing-languages, 2023. Accessed: 2023-01-17.

20 Alastair Green, Paolo Guagliardo, and Leonid Libkin. Property graphs and paths in GQL: Mathematical definitions. Technical Reports TR-2021-01, Linked Data Benchmark Council (LDBC), October 2021. doi:10.54285/ldbc.TZJP7279.

21 Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. Updating graph databases with Cypher. *Proc. VLDB Endow.*, 12(12):2242–2253, 2019. doi:10.14778/3352063.3352139.

22 Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.*, 11(1):27–39, 2017. doi:10.14778/3151113.3151116.

23 Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. Sparqling Kleene: Fast property paths in RDF-3X. In *GRADES*. CWI/ACM, 2013. doi:10.1145/2484425.2484443.

24 A. Gupta and I.S. Mumick. *Materialized Views: Techniques, Implementations, and Applications.* MIT Press, 1999.

25 Aidan Hogan, Cristian Riveros, Carlos Rojas, and Adrián Soto. A worst-case optimal join algorithm for SPARQL. In *ISWC (1)*, volume 11778 of *Lecture Notes in Computer Science*, pages 258–275. Springer, 2019. doi:10.1007/978-3-030-30793-6_15.

26 Egor V. Kostylev, Juan L. Reutter, and Domagoj Vrgoc. Containment of queries for graphs with data. *J. Comput. Syst. Sci.*, 92:65–91, 2018. doi:10.1016/j.jcss.2017.09.005.

27 Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016. doi:10.1145/2850413.

28 Wim Martens, Matthias Niewerth, Tina Popp, Stijn Vansummeren, and Domagoj Vrgoč. Representing paths in graph database pattern matching, 2022. arXiv:2207.13541.

29 Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. Querying the world wide web. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 80–91. IEEE Computer Society, 1996. doi:10.1109/PDIS.1996.568671.

30 Alberto O. Mendelzon and Peter T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995. doi:10.1137/S009753979122370X.

31 Dung T. Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join processing for graph patterns: An old dog with new tricks. In *GRADES*, pages 2:1–2:8. ACM, 2015. doi:10.1145/2764947.2764948.

32 Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. *Theory Comput. Syst.*, 61(1):31–83, 2017. doi:10.1007/s00224-016-9676-2.

33 Marko A. Rodriguez. The Gremlin graph traversal machine and language. In *DBPL*, pages 1–10. ACM, 2015. doi:10.1145/2815072.2815073.

34 Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: a property graph query language. In *GRADES*, page 7. ACM, 2016. doi:10.1145/2960414.2960421.

35 Domagoj Vrgoč. Evaluating regular path queries under the all-shortest paths semantics, 2022. arXiv:2204.11137.

36 Wikipedia contributors. GQL graph query language, 2020. URL: https://en.wikipedia.org/wiki/GQL_Graph_Query_Language.

37 Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Query planning for evaluating SPARQL property paths. In *SIGMOD Conference*, pages 1875–1889. ACM, 2016. doi:10.1145/2882903.2882944.