Querying Graph Data: Where We Are and Where To Go

Leonid Libkin
RelationalAI and IRIF
University of Edinburgh
Paris/Edinburgh, France/UK
l@libk.in

Wim Martens
University of Bayreuth
Bayreuth, Germany
wim.martens@uni-bayreuth.de

Filip Murlak
University of Warsaw
Warsaw, Poland
f.murlak@uw.edu.pl

Liat Peterfreund
Hebrew University of Jerusalem
Jerusalem, Israel
liat.peterfreund@mail.huji.ac.il

Domagoj Vrgoč
Pontificia Universidad Católica de Chile
Santiago, Chile
vrdomagoj@uc.cl

Abstract

Although graph query languages such as Cypher, SQL/PGQ, and GQL take inspiration from theoretical languages such as conjunctive regular path queries (CRPQs), their pattern matching facilities are significantly more powerful in order to cope with real world use cases. Four such extensions are treatment of both nodes and edges, variables that bind to paths or lists, path modes, and data filters.

In this paper, we define *CRPQs with data tests and list variables (dl-CRPQs)*, which extend CRPQs with these features and give the reader a quick idea of how these features relate to the classical literature on graph pattern matching. Then, we discuss where the design of SQL/PGQ and GQL stands today and identify a host of opportunities in research and query language design. In particular, we believe that a closer connection between graph query languages and automata theory will open up opportunities for query optimization that will benefit graph query languages in the long term.

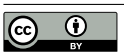
CCS Concepts

• **Information systems** → **Query languages**; **Graph-based database models**; • **Theory of computation** → Formal languages and automata theory.

Keywords

Graph databases, query languages, query language design, regular path queries, paths, automata

ACM Reference Format:
Leonid Libkin, Wim Martens, Filip Murlak, Liat Peterfreund, and Domagoj Vrgoč. 2025. Querying Graph Data: Where We Are and Where To Go. In *Companion of the 44th Symposium on Principles of Database Systems (PODS Companion '25), June 22–27, 2025, Berlin, Germany*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3722234.3725822



PODS Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1565-5/2025/06
https://doi.org/10.1145/3722234.3725822

1 Introduction

"The path is the goal."

— *Ancient wisdom*[1]

Graph query languages have evolved significantly in the last decade. Languages such as Cypher [52, 91], SQL/PGQ [35, 67], and GQL [35, 68] are strongly inspired by conjunctive regular path queries [29], STRUQL [43], GXPath [78], and regular queries [97], and have at the same time taken this academic work significantly further in order to address industrial needs. The pattern matching facilities in these languages extend the classical conjunctive regular path queries (CRPQs) with four main features:

(a) handling both nodes and edges,
(b) path and list variables,
(c) path modes, and
(d) data filters.

The treatment of both nodes and edges is necessary in order to deal with real-world use cases. Since, in graph-structured data, entities are typically modeled as nodes and relationships as edges, a full-fledged query language should be able to query both types of data. Path variables enable returning arbitrary-length paths that match regular path queries, while list variables (called group variables in SQL/PGQ and GQL) allow returning lists of nodes or edges on these paths. These are necessary to let the user inspect the connections that regular path queries find in the data. Path modes (simple, trail, shortest, etc.) were introduced to ensure that the number of paths that matches a regular path query is always finite. Finally, data filters are used for selecting paths based on the data values along them. They play a role similar to the `WHERE` condition in SQL queries.

The design process of Cypher, GQL, and SQL/PGQ has taken academic input into account [5, 6] and has truly taken graph query languages to the next level. Furthermore, these languages are reviving the research interest in query language aspects that were introduced almost 40 years ago [29, 31], leading to new work on combining data and graph topology [78], path modes [13, 83, 85, 87], path variables [41, 84], and formal semantics of query languages [50–52, 56]. On the other hand, seamlessly incorporating all features (a–d) in an industry-level standard is a huge project, for which the current versions of these languages do not mark the end of the road.

[1] It has been attributed to Confucius, Gautama Buddha, Ghandi, and Wim's wife, who regularly reminded him that, when you have small kids, you should focus on the *journey*.

So, how do we study what these languages can and cannot do and, perhaps more importantly, how do we study what they should evolve into? For this, we have two kinds of tools at our disposal. A first kind of tool is abstractions of Cypher, SQL/PGQ, and GQL [50–52, 56]. We refer to such abstractions as being *distilled from practice*. They model the intricacies of these languages fairly precisely, are not always easy to grasp, but are an excellent tool to identify and study potential points of improvement of the original languages. However, being fairly true to the original languages, they necessarily suffer from similar design deficiencies and are not suitable for subjecting the features (a)–(d) to a principled study.

For a principled study of (a)–(d), we need different models that would, ideally, be compatible with the models we have been studying for decades (RPQs, CRPQs, etc.). We call such abstractions *grown from theory*. In this spirit, we introduce *conjunctive regular path queries with data tests and list variables (dl-CRPQs)*, which we believe capture the essence of CRPQs that are extended with features (a)–(d). Ideally, dl-CRPQs can even serve as an inspiration for future graph query languages, since they solve some issues of Cypher, SQL/PGQ, and GQL. Important design principles for dl-CRPQs are:

(1) compatibility with automata,
(2) set semantics, and
(3) symmetry in the treatment of nodes and edges.

It is an interesting question if sticking to these principles could improve the design of real-world graph query languages. We believe so and present a few reasons in the paper.

Some Design Challenges

Queries in SQL/PGQ and GQL, as in Cypher, are built using an intuitive syntax often referred to as "ASCII art". It uses round brackets to denote nodes and square brackets with arrows to denote edges. For example, `(x)` denotes a node which is bound to the variable x, while `-[z:a]->` denotes an a-labeled edge which is bound to the variable z. Such expressions can then be combined freely to define larger patterns. In general, SQL/PGQ and GQL allow treating query patterns as if they were letters in a regular expression (with some syntactic restrictions), thus achieving some degree of syntactic compositionality. Nonetheless, it is not as seamless as compositionality of regular expressions, as we see next. We discuss a few example queries in SQL/PGQ and GQL that highlight some behavior that we believe warrants revisiting certain aspects of their standards.

Example 1. Consider the GQL pattern

```
(x) ( ()-[z:a]->() ){2} (y)
```

It uses variables x and y for nodes and z for edges. Each occurrence of `()` stands for an anonymous variable. The curly brackets `{2}` denote a two-fold iteration. Round brackets are also used for delimiting subexpressions, i.e., to indicate the scope of the iteration operator. The whole pattern matches an a-labeled path of length two from some node x to some node y. Furthermore, to the variable z it associates a list consisting of the two a-labeled edges. The use of lists is convenient because they can be processed later in the query for further filtering, or returned to the user. The pattern, however, is *not equivalent* to any of the following patterns:

```
(x) ( ()-[z:a]->() ) ( ()-[z:a]->() ) (y)
(x) ( ()-[z:a]->()-[z:a]->() ) (y)
(x) ( ()-[z:a]->() ) ( ()-[z1:a]->() ) (y)
```

This is because of the treatment of variables. In each pattern, multiple occurrences of the same variable are interpreted as a join. Therefore in the first two patterns, both occurrences of z match the same edge, so they are not equivalent to the original one. (In fact, both will only match a self-loop, because consecutive node variables, such as `(u)(v)` or `()()`, must match the same node, making the first two patterns equivalent.) The third pattern does match a path of length two from x to y, but instead of associating a list of two elements to z it generates separate bindings for z and z_1. ◀

Example 1 shows a disconnect between SQL/PGQ and GQL patterns and regular expressions, where a^2 is equivalent to aa. We believe that the underlying reason for this disconnect is that variables are used for two rather different purposes: indicating joins and collecting lists of graph elements. The next example shows that these two roles are deeply intertwined.

Example 2. Consider the GQL pattern

```
( (x)-[:a]->(x)-[:a]->() )*
```

The expression uses both occurrences of variable x in two different roles: as a variable to join on and as a variable that produces a list. In order to understand these different roles, it helps to think about the parse tree of the expression. When we see the leaf `(x)`, it is a *node variable*, which means that it binds x to a node. Multiple occurences of the same node variable in a subexpression that does not have iteration are joined, which means that the subexpression `(x)-[:a]->(x)` matches nodes x that have an a-labeled self-loop. However, if in the parse tree we go up through an *iteration* (i.e., repetitions of the form `{a}`, `{a,b}`, Kleene star, or Kleene plus), all the variables in the subexpression become *group variables*. Group variables do not perform joins, but collect in a list all graph elements that are bound to them. Overall, the entire pattern binds x to a *list* of nodes that are connected with a-labeled edges, and in which each node has an a-labeled self-loop. ◀

The example shows that two aspects are closely interleaved: joins (node variables) and creating lists (group variables). From an automata perspective, these two things are very different. Here, creating lists would correspond to *annotating positions* and joins to *registers*. Concerning annotating positions, the community developed a framework with favorable computational complexity properties in the context of *document spanners* [38, 40, 98]. Registers, however, are a feature that drastically changes the power and the computational complexity of basic automata tasks. An automata-aware language design would provide more separation between the roles of node and group variables.

Automata-based approaches *are your friend* when dealing with regular expression matching tasks. For example, huge numbers of matching paths can be stored efficiently [84] by an automata-theoretic variant of database factorization [14, 72, 92]. Furthermore, GQL patterns (without the internal variables we have here) can be evaluated efficiently by traversing the graph and an automaton for the expression in parallel [22, 41]. It is known that such methods lead to algorithms with optimality guarantees [25, 71], as well as unlocking a host of query optimization methods [23, 36, 45].

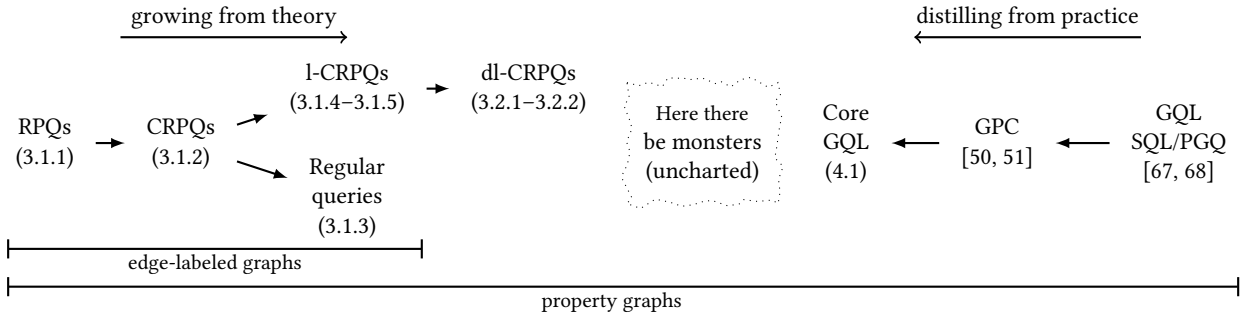


Figure 1: Overview of languages discussed in the paper (the numbers refer to sections or citations)

One of our main goals is to demonstrate that the issues in Example 1 and 2 are solvable: Section 3.1.5 presents a model of RPQs with list variables that is compatible with the aforementioned automata techniques. In Section 3.2.2, this model is extended with joins on node and edge variables, path modes, and data filters. As a final example, we illustrate the importance of symmetry.

Example 3. Consider the GQL pattern

```
(x) ( (u)-[:a]->(v) WHERE u.date < v.date)* (y)
```

This pattern matches a-labeled paths from x to y in which the value of the `date` property of nodes on the path is increasing. It is much less obvious how to match paths in which the value of the `date` property on *edges* is increasing. A naïve approach

```
(x) ( ()-[u:a]->()-[v:a]->() WHERE u.date < v.date)* (y)
```

does not work: it will match a four-edge path in which the values of `date` are 03-01-2025, 04-01-2025, 01-01-2025, 02-01-2025, in this precise order. In fact the only way offered by the language at this point is to match *all paths* and then eliminate those (using `EXCEPT` [68]) in which some two consecutive values of the `date` property are *not* increasing. We revisit this example in more detail in Section 5.2. ◀

Example 3 shows how asymmetric treatment of nodes and edges leads to cases in which a natural condition (increasing dates) can be easily expressed for nodes but not for edges. The asymmetry is that paths in SQL/PGQ and GQL need to start and end with nodes (not with edges), and that neighboring node variables in patterns lead to joins (as in Example 1), but neighboring edge variables do not. In Example 21 we show how a more symmetric treatment of nodes and edges makes increasing values in edges easy to express.

A Zoo of Graph Query Languages

This paper will deal with several languages, coming from different perspectives. We present an overview in Figure 1. Roughly, we see a division into languages that are rooted in theory (with a strive for mathematical clarity and elegance) and languages that are rooted in practice (standards that strive for addressing industrial concerns, and languages that strive for understanding the standards).

The overview just contains languages that we will touch upon in the paper and is far from complete. It does not mention, e.g., two-way navigation [23, 24], scoping rules for path and list variables [50], post-processing of lists and paths [57], due to our focus on specific aspects in modern standards. We refer to Barceló [12] for a more complete overview up to 2013.

The Great 'Relations vs Graphs' Debate

Relational databases have been the foundation of database technology since its very beginning. Historically, we have seen a multitude of alternative database architectures come around [102] and around [103] but, historically, these other architectures have either disappeared or been absorbed by relational systems. So what about graphs?

Here, it is important to distinguish between graph query languages and graph database implementations. We focus on graph query languages and ask questions such as: what kind of features do the users need to express their graph queries easily? This is independent from whether the queried graphs are being implemented in a graph-native or relational database. Graph-native systems may come and go, but graph query languages are here to stay.

2 Graph Data Models

The main graph database models today can roughly be divided into two abstractions: *edge-labeled graphs* and *property graphs*.

Edge-Labeled Graphs. These are the abstraction of graph databases that have been studied since the 1980s [31, 89] and have received a lot of attention in our field due to their elegance and simplicity [9, 10, 12, 81, 94, 114]. Usually, their edges are sets of triples (u, a, v), where u and v are *nodes* and a is a *label*. Such a definition is very close to the Resource Description Framework (RDF) format [30], which is rooted in the Semantic Web community. While this model is simple and flexible, an important drawback is its inability to address edges directly: edges do not have identifiers. Indeed, this is a common issue in RDF that led to several extension proposals such as RDF* [63, 64] or multi-layered graphs [7, 66, 75, 110]. We therefore define edge-labeled graphs slightly more generally, bringing them closer to the property graph model (Definition 6). In the following we assume disjoint countably infinite sets Nodes of node identifiers, Edges of edge identifiers, and Labels of edge labels.

Definition 4. An *edge-labeled graph* G is a tuple $(N, E, \mathrm{src}, \mathrm{tgt}, \lambda)$, where

- $N \subseteq$ Nodes is a finite set of nodes,
- $E \subseteq$ Edges is a finite set of edges,
- $\mathrm{src} : E \to V$ is a total function,
- $\mathrm{tgt} : E \to V$ is a total function, and
- $\lambda : E \to$ Labels is a total function assigning labels to edges.

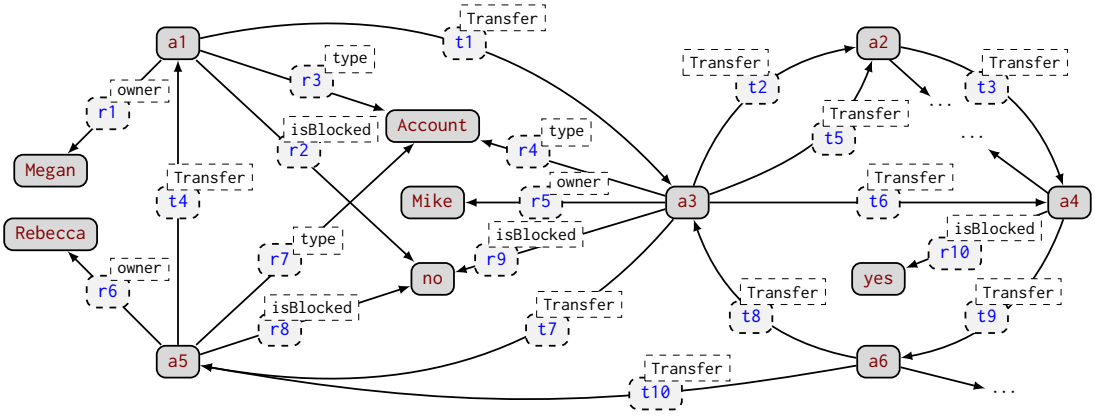


Figure 2: An edge-labeled graph with bank accounts (a1–a6) and transfers (t1–t10) between them.

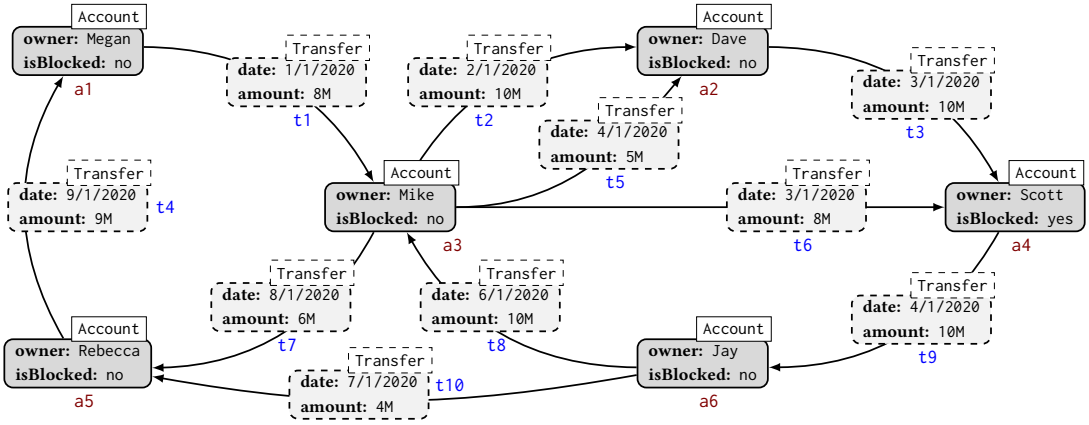


Figure 3: A property graph with information on bank accounts and money transfers.

Intuitively, $\mathrm{src}(e) = n_1$ and $\mathrm{tgt}(e) = n_2$ means that e is a directed edge going from n_1 to n_2, while $\lambda(e) = a$ means that e is labeled with a. In contrast to a set of triples of the form (n_1, a, n_2), this model can have different edges e_1, e_2 going from n_1 to n_2 having the label a.

Example 5. Figure 2 partially shows an example of an edge-labeled graph. (It does not convey the same information as the property graph in Figure 3; some nodes and edges are omitted to avoid cluttering). Its nodes are {a1, . . . , a6, Account, Megan, Mike, Rebecca, no, . . . } and its edges are {t1, . . . , t10, r1, . . . } (nodes are solid, edges are dashed). Furthermore, we have $\lambda($t1$) =$ Transfer, $\lambda($r1$) =$ owner, etc. Notice that edges t2 and t5 are both from a3 to a2 and both have the label Transfer.

Labeled Property Graphs. This is the model deployed in many enterprise systems [91, 93, 105, 106, 109] and it extends edge-labeled graphs by adding attributes (properties) with their associated values to both nodes and edges, as well as allowing to put labels on nodes. For this, in addition to Nodes, Edges, and Labels, we now assume a countably infinite set Properties of node and edge property names and Values of property values. Compared to relational databases, property names and values correspond to attribute names and values.

Definition 6. A *labeled property graph (LPG)* is a tuple $G = (N, E, \mathrm{src}, \mathrm{tgt}, \lambda, \rho)$, where

- $N \subseteq$ Nodes is a finite set of nodes,
- $E \subseteq$ Edges is a finite set of edges,
- $\mathrm{src} : E \to V$ is a total function,
- $\mathrm{tgt} : E \to V$ is a total function,
- $\lambda : (N \cup E) \to$ Labels is a total function assigning a label to an edge or a node, and
- $\rho : (N \cup E) \times$ Properties $\to$ Values is a partial function.

Here $\rho(x, \mathrm{prop}) = v$ means that the node or edge x has the property prop whose value is v. Given that ρ is a partial function not every node and edge has to have a value for each property, nor do they have to have the same properties with defined values. In the remainder of the paper, we will usually omit "labeled" when referring to labeled property graphs.

Notice that, if $(N, E, \mathrm{src}, \mathrm{tgt}, \lambda, \rho)$ is a property graph, then $(N, E, \mathrm{src}, \mathrm{tgt}, \lambda|_E)$ is an edge-labeled graph. Here, we denote by $\lambda|_E$ the restriction of λ to the domain E.

Remark 7. *Definition 6 allows a single label per node or edge. In some practical languages, multiple labels are allowed. This omission makes no difference in terms of the results presented in this paper.*

Example 8. Figure 3 depicts a property graph with nodes {a1, ..., a6} and edges {t1, ..., t10}. Similar to Figure 3, nodes are solid and edges are dashed. Concerning labels, we have $\lambda(\texttt{a1}) = \texttt{Account}$, $\lambda(\texttt{t1}) = \texttt{Transfer}$, etc. Concerning properties and values, we have $\rho(\texttt{a1}, \texttt{owner}) = \texttt{Megan}$, etc.

Paths and Lists. A *path* in a graph is an alternating sequence of nodes and edges such that consecutive elements are incident. We consider four types of paths, depending on whether they begin with a node or and edge; or end with a node or an edge: *node-to-node* paths, *node-to-edge* paths, etc. We write paths as

$$p = \mathsf{path}(o_1, \ldots, o_n)$$

where the o_i with $i \in [n]$ can be nodes or edges. GQL and SQL/PGQ use the term *elements* for nodes or edges. We use the term *objects* and denote them with $o, o_i, \ldots$ to avoid notation clashes with edges, which we denote with $e, e_i, \ldots$. We write path() for the empty path. For a non-empty path p, by $\mathsf{src}(p)$ we mean o_1 if o_1 is a node, and $\mathsf{src}(o_1)$ if o_1 is an edge. Similarly $\mathsf{tgt}(p)$ is o_n if o_n is a node and $\mathsf{tgt}(o_n)$ if it is an edge. We call p a path *from* $\mathsf{src}(p)$ *to* $\mathsf{tgt}(p)$. We write $\mathsf{len}(p)$ for the *length* of p, which we define as its number of edges; that is, $\mathsf{len}(p) = |\{i \mid o_i \text{ is an edge}\}|$. Notice that edges that appear multiple times are also counted multiple times.

Let G be an edge-labeled graph or a property graph with functions src, tgt, and λ. We say that p is a *path in* G if each edge in p connects the nodes before and after it in the sequence; that is,

(a) for each $i \in \{1, \ldots, n-1\}$ such that o_i is an edge, we have that $\mathsf{tgt}(o_i) = o_{i+1}$ and
(b) for each $i \in \{2, \ldots, n\}$ such that o_i is an edge, we have that $\mathsf{src}(o_i) = o_{i-1}$.

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

Remark 9. *While practical languages [35, 52, 112] use two-way paths with forward and backward edges, we focus on one-way paths. This is just for the sake of technical simplicity: our framework can easily be extended with two-way paths.*

To handle the fact that our paths can begin or end in either a node or an edge, we define path concatenation as follows. For $p = \mathsf{path}(o_1, \ldots, o_n)$ and $p' = \mathsf{path}(o'_1, \ldots, o'_m)$ we define $p \cdot p'$ to be the path

- $\mathsf{path}(o_1, \ldots, o_n, o'_1, \ldots, o'_m)$ if $o_n \in E$ and $\mathsf{tgt}(o_n) = o'_1$;
- $\mathsf{path}(o_1, \ldots, o_n, o'_1, \ldots, o'_m)$ if $o'_1 \in E$ and $\mathsf{src}(o'_1) = o_n$; and
- $\mathsf{path}(o_1, \ldots, o_n, o'_2, \ldots, o'_m)$ if $o_n = o'_1$.

Furthermore, we define $p \cdot \mathsf{path}() = p = \mathsf{path}() \cdot p$. Notice that $\mathsf{path}(o) \cdot \mathsf{path}(o) = \mathsf{path}(o)$, independent of whether o is a node or an edge. This is different from the GQL standard in which this equality holds for nodes but not for edges, but it helps to arrive at more elegant definitions for (conjunctive) regular path queries with data value comparisons and list variables (Sections 3.2.1–3.2.2).

Example 10. Consider $\mathsf{path}(\texttt{a1}, \texttt{t1}, \texttt{a3}, \texttt{t2})$, which is a valid path in the property graph of Figure 3. This is an example of a node-to-edge path, since it starts with a node and ends with an edge. Similarly, $\mathsf{path}(\texttt{t1}, \texttt{a3}, \texttt{t2})$ is a valid edge-to-edge path. However, expressions such as $\mathsf{path}(\texttt{a1}, \texttt{t1}, \texttt{t1})$ do not define a valid path since they repeat an edge without interleaving it with a node.

Given that we have to take care of different start/end objects, the concatenation is also more involved. For example, the path $\mathsf{path}(\texttt{a1}, \texttt{t1}, \texttt{a3}, \texttt{t2}, \texttt{a2})$ in the graph of Figure 3 can be expressed as a concatenation of two paths in several different ways, including:

$$\begin{gathered}\mathsf{path}(\texttt{a1}, \texttt{t1}, \texttt{a3}) \cdot \mathsf{path}(\texttt{a3}, \texttt{t2}, \texttt{a2}) \\ \mathsf{path}(\texttt{a1}, \texttt{t1}) \cdot \mathsf{path}(\texttt{a3}, \texttt{t2}, \texttt{a2}) \\ \mathsf{path}(\texttt{a1}, \texttt{t1}) \cdot \mathsf{path}(\texttt{t1}, \texttt{a3}, \texttt{t2}, \texttt{a2}) \\ \ldots\end{gathered}$$

Notice that the third concatenation above shows that the length of the path obtained by concatenating two paths is not necessarily the sum of the length of concatenated paths. This happens since the repeated edge gets collapsed into a single edge as in the example above. Our design decision is to prioritize symmetric treatment of nodes and edges over requiring the length of a concatenated path to be the sum of the length of the parts, which will lead to much more symmetry between treatment of nodes and edges in Section 3.2.1 compared to Cypher, SQL/PGQ, and GQL (and, in particular, in Example 21).

For this reason, if we want to obtain a path that traverses a self-loop twice by concatenating two paths that only consist of the edge, we need to consider the incident node. Indeed, if we assumed that there is a self-loop t0 over the node a1, then we have that $\mathsf{path}(\texttt{t0}) \cdot \mathsf{path}(\texttt{t0}) = \mathsf{path}(\texttt{t0})$, whereas $\mathsf{path}(\texttt{t0}) \cdot \mathsf{path}(\texttt{a1}, \texttt{t0}) = \mathsf{path}(\texttt{t0}, \texttt{a1}, \texttt{t0})$, which is the path that traverses the edge t0 twice.

We define the *edge label* of p, denoted $\mathsf{elab}(p)$, inductively as

$$\mathsf{elab}(o) = \begin{cases} \lambda(o) & \text{if } o \text{ is an edge} \\ \varepsilon & \text{if } o \text{ is a node.} \end{cases}$$

Furthermore, $\mathsf{elab}(p_1 \cdot p_2) = \mathsf{elab}(p_1) \cdot \mathsf{elab}(p_2)$.

Some of the graph query languages in Section 3.1 use lists. We write lists as $\mathsf{list}(o_1, \ldots, o_n)$, where $o_1, \ldots, o_n$ is the sequence of elements in the list. We write list() for the empty list. The concatenation of two lists $\mathsf{list}(o_1, \ldots, o_n)$ and $\mathsf{list}(o'_1, \ldots, o'_m)$, written as $\mathsf{list}(o_1, \ldots, o_n) \cdot \mathsf{list}(o'_1, \ldots, o'_m)$ is $\mathsf{list}(o_1, \ldots, o_n, o'_1, \ldots, o'_m)$.

Variables and their Bindings. To define our query languages, we assume that we have a countably infinite set Var of variables that we use for nodes, edges, or even lists of nodes and/or edges; and DataVar of *data variables*, which we assume to be disjoint from Var. We will also use *bindings*, which are partial mappings from Var to lists of edges, and nodes. The image of a binding depends on the query languages that we define.

3 Query Languages Rooted in Theory

We present a simple formal model of today's graph query languages in several steps, starting from the ubiquitous *regular path queries*. We aim at a model that is theoretically elegant, at the cost of not being completely faithful to the intricate details of Cypher, GQL, or SQL/PGQ, much like what first-order logic is to SQL. For more faithful abstractions of Cypher, GQL, and SQL/PGQ, see [50–52].

3.1 Querying Edge-Labeled Graphs

We start with the (conjunctive) regular path queries that are rooted in the work of Cruz et al. from 1987 [31]. In Sections 3.1.4–3.1.5, we extend these with *list variables* that can bind to elements of paths. They abstract the *group variables* in Cypher, GQL, and SQL/PGQ.

3.1.1 Regular Path Queries. A *regular path query (RPQ)* is a regular expression R over Labels. The regular expressions we consider are inductively defined as the smallest set that contains ε (empty base case), every element of Labels (label base case) and, if R_1 and R_2 are regular expressions, then so are their *concatenation* $(R_1 \cdot R_2)$, *disjunction* $(R_1 + R_2)$, and the *Kleene star* R_1^*. The language $L(R)$ of a regular expression R is defined as usual and we use the usual rules for omitting brackets and concatenation symbols. For convenience, we use $R?$ to abbreviate $R + \varepsilon$ and R^+ to abbreviate $R \cdot R^*$.

Remark 11 (Wildcards). *Real-world graph database query languages typically have wildcards that allow a single position of an RPQ to match an infinite subset of* Labels. *Cypher, GQL, and SQL/PGQ use anonymous variables, denoted* `()` *for nodes and* `[]` *for edges. We will sometimes use wildcards later in the paper in the same way as they are used in SPARQL. Here, we allow regular expression base expressions of the form* $!S$*, where* $S \subseteq$ Labels *is finite. Their semantics is* $L(!S) = \{a \in \text{Labels} \mid a \notin S\}$*. Our reason for using this kind of wildcards is that they allow translations to finite automata and standard automata constructions such as union, intersection, determinization, and complement. Extending later definitions in the paper with such wildcards is routine. We sometimes use the notation "_" for* $(a + !\{a\})$*, which matches every* $a \in$ Labels.

Let $G = (N, E, \text{src}, \text{tgt}, \lambda)$ be an edge-labeled graph. We denote the *result of* R *on* G as $[\![R]\!]_G$ and define it as

$$[\![R]\!]_G = \big\{(u, v) \in N \times N \mid \text{there is a path } p \text{ from } u \text{ to } v \text{ in } G \text{ with } \text{elab}(p) \in L(R)\big\}\,.$$

Example 12. Consider the RPQ $R = \text{Transfer}^*$. On the graph in Figure 2, it returns the complete set of pairs $\{\text{a1}, \ldots, \text{a6}\} \times \{\text{a1}, \ldots, \text{a6}\}$, since all these nodes are strongly connected with Transfer-edges.

3.1.2 Conjunctive Regular Path Queries. A *conjunctive regular path query (CRPQ)* is an expression of the form

$$q(x_1, \ldots, x_k) :- R_1(y_1, y_1'), \ldots, R_n(y_n, y_n')$$

where

(1) R_i is a regular path query for each $i \in [n]$,
(2) $\{x_1, \ldots, x_k, y_1, y_1', \ldots, y_n, y_n'\} \subseteq \text{Var}$, and
(3) $\{x_1, \ldots, x_k\} \subseteq \{y_1, y_1', \ldots, y_n, y_n'\}$.

We sometimes write the entire query as q for simplicity. To define its semantics on an edge-labeled graph $G = (N, E, \text{src}, \text{tgt}, \lambda)$, we say that a *node homomorphism from* q *to* G is a mapping $h : \{y_1, y_1', \ldots, y_n, y_n'\} \to N$ such that $(h(y_i), h(y_i')) \in [\![R_i]\!]_G$ for every $i \in [n]$. The output of q on G is then defined as

$$q(G) = \{h(x_1, \ldots, x_k) \mid h \text{ is a node homomorphism from } q \text{ to } G\}\,.$$

Here, we write the tuple $(h(x_1), \ldots, h(x_k))$ as $h(x_1, \ldots, x_k)$.

Example 13. The CRPQ

$$q_1(x_1, x_2, x_3) :- \texttt{Transfer}(x_1, x_2), \texttt{Transfer}(x_1, x_3), \texttt{Transfer}(x_2, x_3)$$

returns $\{(\text{a3}, \text{a2}, \text{a4}), (\text{a6}, \text{a3}, \text{a5})\}$ on the graph in Figure 2. The CRPQ

$$q_2(x, x_1, x_2) :- \texttt{owner}(y, x_1), \texttt{isBlocked}(y, x_2), (\texttt{Transfer} \cdot \texttt{Transfer?} \cdot \texttt{Transfer?})(x, y)$$

matches accounts x and y such that there is a path of transfers from x to y of length 1 to 3. It then returns triples consisting of account x, the owner of y, and whether the account y is blocked. One such example in Figure 2 is (a4, Rebecca, no). Indeed, there is a path of transfers of length 2 from a4 to a5.

3.1.3 Nesting queries: a side trip. Given that a binary query returns pairs of nodes, conceptually we can think of such a query as defining virtual edges. A *compositional* query language should then have full access to these virtual edges which are defined by queries from the language. Owing to the closure properties of regular expressions, RPQs are compositional in this sense, but CRPQs are not. Indeed, in CRPQs it is not possible to take transitive (Kleene) closures of results of CRPQs. We illustrate this in the next example.

Example 14. Consider a simple CRPQ

$$q_1(x, y) :- \texttt{Transfer}(x, y), \texttt{Transfer}(y, x)\,,$$

which finds pairs of nodes connected by `Transfer`-labeled edges in both directions. If we think of q_1 as defining (virtual) edges, it is then not possible to construct a CRPQ that would select pairs of nodes connected by a path of such edges.

This kind of compositionality can be achieved by nesting queries, an idea already present in [29] and further developed in [21] and [97]. In essence, the idea is to allow using binary CRPQs in place of edge labels in RPQs. The following example illustrates this.

Example 15. Pairs of nodes connected by a path of virtual edges defined by q_1 from Example 14 can be selected using a nested CRPQ:

$$q_2(u, v) :- \big((\texttt{Transfer}(x, y), \texttt{Transfer}(y, x))[x, y]\big)^*(u, v)\,.$$

We call such queries *nested CRPQs*. Reutter et al. [97] introduced an elegant Datalog-like syntax for nested CRPQs and coined the term *regular queries*. GQL and SQL/PGQ follow this idea closely and allow applying Kleene star to every pattern. While in this paper we focus on "flat" CRPQs, nesting is a powerful feature of real-life graph query languages that deserves more attention.

3.1.4 RPQs with List Variables. An *RPQ with list variables (l-RPQ)* is a regular expression R over $\text{Labels} \cup \{a^z \mid a \in \text{Labels}, z \in \text{Var}\}$, where we assume the union to be disjoint. Intuitively, the elements of Labels work the same as before, and elements of the form a^z match an edge with label a and add their ID to a list associated to the variable $z \in \text{Var}$. In order to avoid confusion, we use letters x, y for "CRPQ variables" and z for list variables. By $\text{Var}(R)$ we denote the set of variables occurring in R.

We formalize their semantics using *path bindings*, which are pairs (p, μ) where p is a path and μ is a binding that maps variables in Var to lists of edges that appear in p. We assume the bindings μ to be total on Var, but only map to non-empty lists for a finite number of variables. This facilitates the definition of their concatenation: for two bindings μ_1 and μ_2, we denote by $\mu_1 \cdot \mu_2$ the binding that

concatenates all lists. That is, for every $z \in \mathrm{Var}$, we define $(\mu_1 \cdot \mu_2)(z) = \mu_1(z) \cdot \mu_2(z)$. By μ_0 we denote the binding such that $\mu_0(z) = \mathrm{list}()$ for every $z \in \mathrm{Var}$. By $\mu_{z \mapsto e}$ we denote the binding μ such that $\mu(z) = \mathrm{list}(e)$ and $\mu(z') = \mathrm{list}()$ for every $z' \neq z$. For an edge-labeled graph $G = (N, E, \mathrm{src}, \mathrm{tgt}, \lambda)$, we define:

$$[\![\varepsilon]\!]_G := \{(\mathrm{path}(u), \mu_0) \mid u \in N\}$$
$$[\![a]\!]_G := \{(\mathrm{path}(u, e, v), \mu_0) \mid e \in E, \mathrm{src}(e) = u, \mathrm{tgt}(e) = v, \lambda(e) = a\}$$
$$[\![a^z]\!]_G := \{(\mathrm{path}(u, e, v), \mu_{z \mapsto e}) \mid e \in E, \mathrm{src}(e) = u, \mathrm{tgt}(e) = v, \lambda(e) = a\}$$
$$[\![R_1 + R_2]\!]_G := [\![R_1]\!]_G \cup [\![R_2]\!]_G$$
$$[\![R_1 \cdot R_2]\!]_G := \{(p_1 \cdot p_2, \mu_1 \cdot \mu_2) \mid (p_1, \mu_1) \in [\![R_1]\!]_G, (p_2, \mu_2) \in [\![R_2]\!]_G\}$$
$$[\![R]\!]_G^0 := \{(\mathrm{path}(u), \mu_0) \mid u \in N\}$$
$$[\![R]\!]_G^k := \{(p_1 \cdot \ldots \cdot p_k, \mu_1 \cdot \ldots \cdot \mu_k) \mid (p_i, \mu_i) \in [\![R]\!]_G \text{ for all } i \in [k]\}$$
$$[\![R^*]\!]_G := \bigcup_{k=0}^{\infty} [\![R]\!]_G^k$$

Example 16. The l-RPQ

$$R = (\texttt{Transfer}^z)^* \cdot \texttt{isBlocked}$$

matches paths consisting of arbitrarily many `Transfer`-edges and ending with an `isBlocked`-edge. Furthermore, it stores the list of `Transfer`-edges in variable z. On the graph G in Figure 2, it returns the infinite set $[\![R]\!]_G = \{(\mathrm{path}(\texttt{a4}, \texttt{r10}, \texttt{yes}), \mu_1), (\mathrm{path}(\texttt{a2}, \texttt{t3}, \texttt{a4}, \texttt{r10}, \texttt{yes}), \mu_2), (\mathrm{path}(\texttt{a3}, \texttt{t2}, \texttt{a2}, \texttt{t3}, \texttt{a4}, \texttt{r10}, \texttt{yes}), \mu_3), \mathrm{path}(\texttt{a3}, \texttt{t5}, \texttt{a2}, \texttt{t3}, \texttt{a4}, \texttt{r10}, \texttt{yes}), \mu_4), \ldots, \mathrm{path}(\texttt{a3}, \texttt{r9}, \texttt{no}), \mu_5), \ldots\}$, where

$$\mu_1(z) = \mathrm{list}() \qquad \mu_2(z) = \mathrm{list}(\texttt{t3}) \qquad \mu_3(z) = \mathrm{list}(\texttt{t2}, \texttt{t3})$$
$$\mu_4(z) = \mathrm{list}(\texttt{t5}, \texttt{t3}) \qquad \mu_5(z) = \mathrm{list}()$$

Notice that, since we use edge identity in edge-labeled graphs (Definition 4), we can now differentiate between returning the "parallel edges" `t2` or `t5`.

The example illustrated that $[\![R]\!]_G$ can be infinite, just as for regular expressions. For database queries, however, we typically want to have finite outputs. For this reason, GQL and SQL/PGQ have the ability to restrict paths to shortest, simple, or trails, which always ensures a finite output. For technical reasons, we introduce these restrictions when we move to *conjunctions* of RPQs with list variables (Section 3.1.5). We believe that RPQs with list variables are interesting to study even in the case where $[\![R]\!]_G$ is infinite. For instance, one can study enumeration of all results that match paths of a given length (see e.g., [87]).

Furthermore, l-RPQs are designed to allow a translation into finite automata using routine methods (similar to those used in the research on *document spanners* [2, 38, 40, 98]). Such a design has important advantages:

(1) it allows to leverage factorization for storing sets of paths — even if these sets are infinite [41, 84] (see Section 6); and
(2) it avoids semantic issues caused by the (overly?) syntax-driven design of GQL group variables.

Concerning (2), list variables indeed do not have the semantic issues highlighted in Example 1 in the introduction, because they satisfy $[\![R]\!]_G^2 = [\![R \cdot R]\!]_G$ by definition. On the other hand, they *only* collect graph elements in lists and do not support joins. We believe that joins should happen at the level of CRPQs, which we discuss next.

3.1.5 CRPQs with List Variables.

A *CRPQ with list variables* is an expression of the form

$$q(x_1, \ldots, x_k) :- m_1\, R_1(y_1, y_1'),\ \ldots\ ,\ m_n\, R_n(y_n, y_n')$$

where

(1) $m_i \in \{\mathrm{shortest}, \mathrm{simple}, \mathrm{trail}, \mathrm{all}\}$;
(2) R_i is an RPQ with list variables for each $i \in [n]$;
(3) $\mathrm{Var}(R_i) \cap \{y_1, y_1', \ldots, y_n, y_n'\} = \emptyset$ for every $i \in [n]$;
(4) $\mathrm{Var}(R_i) \cap \mathrm{Var}(R_j) = \emptyset$ for every $i \neq j$ with $i, j \in [n]$; and
(5) $\{x_1, \ldots, x_k\} \subseteq \{y_i, y_1', \ldots, y_n, y_n'\} \cup \bigcup_{i=1}^n \mathrm{Var}(R_i)$.

Recall that the sets $\mathrm{Var}(R_i)$ contain list variables that we denote with z, z_i, etc.

To define the semantics on edge-labeled graphs, we first introduce some notation. For a set P of path bindings over G and nodes u, v of G, we let $\sigma_{u,v}(P) = \{(p, \mu) \in P \mid p \text{ is a path from } u \text{ to } v\}$. Furthermore, we define

- $\mathrm{shortest}(P) = \{(p, \mu) \in P \mid \mathrm{len}(p) \text{ is minimal in } P\}$,
- $\mathrm{simple}(P) = \{(p, \mu) \in P \mid p \text{ is a simple path}\}$,
- $\mathrm{trail}(P) = \{(p, \mu) \in P \mid p \text{ is a trail}\}$,
- $\mathrm{all}(P) = P$.

An *unrestricted path homomorphism* from q to G is a mapping $h : \mathrm{Var}(q) \to N \cup \mathrm{list}(G)$ such that

- $h|_{\{y_1, y_1', \ldots, y_n, y_n'\}}$ is a node homomorphism from q to G, and
- for each $i \in [n]$ and $z \in \mathrm{Var}(R_i)$, we have $h(z) = \mu(z)$, for some
$$(p, \mu) \in \sigma_{h(y_i), h(y_i')}\big([\![R_i]\!]_G\big)\ .$$

A *(restricted) path homomorphism* takes the m_i into account. It is defined analogously, but

- for each $i \in [n]$ and $z \in \mathrm{Var}(R_i)$, we have $h(z) = \mu(z)$, for some
$$(p, \mu) \in m_i\big(\sigma_{h(y_i), h(y_i')}\big([\![R_i]\!]_G\big)\big)\ .$$

The output of q on G is then defined as

$$q(G) = \{h(x_1, \ldots, x_k) \mid h \text{ is a path homomorphism from } q \text{ to } G\}\ .$$

Example 17 (Grouping by Endpoint Pairs). The l-CRPQ

$$q(x_1, x_2, z) :-$$
$$\texttt{owner}(y_1, x_1), \texttt{owner}(y_2, x_2), \mathrm{shortest}\ (\texttt{Transfer}^z)^+(x_1, x_2)$$

returns the owners x_1 and x_2 of accounts y_1 and y_2, respectively, together with the lists of edges in shortest paths of transfers from x_1 to x_2 that have nonzero length. (To simplify notation, we omit the modifiers all.) If we assume for the sake of consistency with Figure 3 that `a6` has an outgoing `owner`-edge to `Jay`, then it returns

$$x_1 \mapsto \texttt{Jay} \qquad x_1 \mapsto \texttt{Mike}$$
$$x_2 \mapsto \texttt{Rebecca} \qquad x_2 \mapsto \texttt{Megan}$$
$$z \mapsto \mathrm{list}(\texttt{t10}) \qquad z \mapsto \mathrm{list}(\texttt{t7}, \texttt{t4})$$

etc. Notice that q returns shortest paths *grouped by endpoint pair* of the l-RPQ $(\texttt{Transfer}^z)^+$. Indeed, for the (infinitely many) paths from `a6` (`Jay`) to `a5` (`Rebecca`), the restricted path homomorphism applies end-node selection before shortest, which is why it selects the path of length one. For the paths from `a3` (`Mike`) to `a1` (`Megan`), following the same principle, it selects the path of length two. This is the desired behavior of the shortest mode in GQL.

Remark 18. *List variables could be added to nested CRPQs (Section 3.1.3) or regular queries in a similar way, but the semantics of* shortest, simple, *and* trail *would have to be reconsidered, because atoms of such queries do not match paths, but more complex structures.*

3.2 Querying Property Graphs

We now equip the languages described in Sections 3.1.4 and 3.1.5 with features to query property graphs: most notably, the capability to query data values. The languages presented here find their roots in query languages for *data graphs* [78], where each node carries a data value from Values. Data graphs were motivated by *data trees*, which were studied in the context of XML and tree-structured data during the decade prior to its appearance [17–20]. Compared to query languages for data graphs, we incorporate the following features to bring the formalism closer to GQL: (1) list variables and (2) the capability to distinguish between nodes and edges. Our languages will deal with nodes and edges in a symmetric way, although this is not the case in SQL/PGQ and GQL.

3.2.1 RPQs with Data Tests and List Variables. Consider the following grammar of *element tests* (denoted ETest for short), defining operators which allow testing and storing property values:

$$\text{ETest} \;:\!-\; x := \text{pname} \;\mid\; \text{pname}\ op\ c \;\mid\; \text{pname}\ op\ x$$

Here, $op \in \{=, \neq, <, >\}$, pname $\in$ Properties, $c \in$ Values, and $x \in$ DataVar is a variable used to bind values from Values.

A *regular expression with data tests and list variables* is an expression of the form $R(x, y)$, where R is a regular expression over

$$\{(a) \mid a \in \text{Labels}\} \cup \{[a] \mid a \in \text{Labels}\} \cup$$
$$\cup \{(a^z) \mid z \in \text{Var}, a \in \text{Labels}\} \cup \{[a^z] \mid z \in \text{Var}, a \in \text{Labels}\} \cup$$
$$\cup \{(\text{et}) \mid \text{et} \in \text{ETest}\} \cup \{[\text{et}] \mid \text{et} \in \text{ETest}\}$$

Intuitively, we use $(\cdot)$ to match nodes and $[\cdot]$ to match edges. An *RPQ with data tests and list variables*, or *dl-RPQ* for short, is an expression of the form $R(x, y)$, where R is a regular expression with data tests and list variables.

Semantics. To define the semantics, we will use *value assignments*, which are partial functions $\nu : \text{Var} \to \text{Values}$. By $\nu[x \mapsto c]$ we denote the mapping that is identical to ν, except that it maps x to c. For a property graph $G = (N, E, \text{src}, \text{tgt}, \lambda, \rho)$, we define the relation

$$(p, \nu, \mu) \vdash_R (p', \nu', \mu')$$

by induction on the structure of the dl-RPQ R. Here, p and p' are paths, ν and ν' are partial mappings from Var to Values, and μ and μ' are bindings that map to lists of graph elements. Recall that we defined $\mu_1 \cdot \mu_2$ and $\mu_{z \mapsto o}$ in Section 3.1.4. We then require the following, depending on the structure of R:

- $(a) : p' = p \cdot \text{path}(o), o \in N, \lambda(o) = a,$ and $(\nu', \mu') = (\nu, \mu)$
- $(a^z) : p' = p \cdot \text{path}(o), o \in N, \lambda(o) = a, \nu' = \nu,$ and $\mu' = \mu \cdot \mu_{z \mapsto o}$
- (et) : $p' = p \cdot \text{path}(o), o \in N$ and, depending on the structure of et,
  - $(x := \text{pname}) : \nu' = \nu[x \mapsto \rho(n, \text{pname})]$,
  - (pname **op** c) : $\rho(o, \text{pname})$ **op** c holds,
  - (pname **op** x) : $\rho(o, \text{pname})$ **op** $\nu(x)$ holds;
- $R_1 \cdot R_2$: there is a (p_1, ν_1, μ_1) such that $(p, \nu, \mu) \vdash_{R_1} (p_1, \nu_1, \mu_1)$ and $(p_1, \nu_1, \mu_1) \vdash_{R_2} (p', \nu', \mu')$
- $R_1 + R_2 : (p, \nu, \mu) \vdash_{R_1} (p', \nu', \mu')$ or $(p, \nu, \mu) \vdash_{R_2} (p', \nu', \mu')$
- $R^* : (p, \nu, \mu) = (p', \nu', \mu')$ or there are $(p_1, \nu_1, \mu_1), \ldots, (p_n, \nu_n, \mu_n)$ such that $(p, \nu, \mu) \vdash_R (p_1, \nu_1, \mu_1), (p_n, \nu_n, \mu_n) \vdash_R (p', \nu', \mu')$, and $(p_i, \nu_i, \mu_i) \vdash_R (p_{i+1}, \nu_{i+1}, \mu_{i+1})$ for every $i \in [n-1]$.

The base cases $[a]$, $[a^z]$, and [et] are analogous, but have the condition $o \in E$ instead of $o \in N$.

Finally, for a dl-RPQ r, we define

$$[\![R]\!]_G = \{(p, \mu) \mid (\text{path}(), \nu_0, \mu_0) \vdash_r (p, \nu, \mu)\},$$

where ν_0 is the empty assignment and μ_0 maps every variable to the empty list (as in Section 3.1.4).

We rely crucially on our definition of path concatenation, which ensures that $p \cdot \text{path}(o) = p$ if the last object of p is o. and thus facilitates matching multiple atomic expressions to the same object. For example, similar to GQL patterns, $(a^z)(\text{date} < x)(x := \text{date})$ will be matched to a single node: first it tests if the current node is labeled a and adds it to the list variable z; then it tests if the value of its date property is smaller than the one stored in x; and then it overwrites x with this value. Different from GQL, we treat nodes and edges symmetrically here, and $[a^z][\text{date} < x][x := \text{date}]$ will also be matched to a single edge.

Remark 19. *Note that we do not include value assignments in the output of dl-RPQs: $[\![R]\!]_G$ is defined as a set of pairs of the form (p, μ). This means that we use value assignments only for filtering paths (much like the WHERE clause in SQL), and not for returning values.*

Remark 20. *Notice that dl-RPQs can express boolean combinations of* ETest*s. Indeed, conjunction is concatenation in regular expressions. Disjunction is disjunction in regular expressions. Negation can be pushed to atoms and then we use that op has $\{=, \neq, <, >\}$.*

Example 21. The following expression selects paths that are labeled a^* and have increasing values of the date property in nodes (recall from Remark 11 that we use "_" to denote a label wildcard):

$$(a^z)(x := \text{date})\Big([\_](a^z)(\text{date} > x)(x := \text{date})\Big)^*$$

The following expression does the analogous thing for edges:

$$[a^z][x := \text{date}]\Big((\_)[a^z][\text{date} > x][x := \text{date}]\Big)^*$$

The latter expression returns edge-to-edge paths. If we would like to have node-to-node paths, we can write

$$(\_)[a^z][x := \text{date}]\Big((\_)[a^z][\text{date} > x][x := \text{date}]\Big)^*(\_)$$

3.2.2 CRPQs with Data Tests and List Variables. A *CRPQ with data tests and list variables*, or *dl-CRPQ* for short, is an expression of the form

$$q(x_1, ..., x_k) :\!-\; m_1\, R_1(y_1, y_1'), \ \ldots\ , \ m_n\, R_n(y_n, y_n')$$

that satisfies conditions (1–5) for CRPQs with list variables (Section 3.1.5), except that $R_i(y_i, y_i')$ can now be dl-RPQs in condition (2). The semantics of dl-CRPQs is verbatim the same as the semantics of CRPQs with list variables in Section 3.1.5.

4 Query Languages: From Practice to Theory

In Section 3, we were building graph languages bottom-up, starting with the basic RPQs and then adding features such as list variables, joins, and data tests (the *growing from theory* part of Figure 1).

We now contrast this with a top-down approach, where we start with a real language and try to distill it into a clean theoretical calculus that captures its essential features. Specifically, we start with GQL, whose standard is 500 pages of dense dry text written in a very specific language mandated by ISO [68], hard to understand

$$
\begin{array}{rcl}
[\![(x)]\!]_G & := & \{(\mathrm{path}(n), \{x \mapsto n\}) \mid n \in N\} \\
\left[\!\!\left[\overset{x}{\rightarrow}\right]\!\!\right]_G & := & \{(\mathrm{path}(n_1, e, n_2), \{x \mapsto e\}) \mid e \in E,\ \mathrm{src}(e) = n_1,\ \mathrm{tgt}(e) = n_2\} \\
[\![\pi_1\, \pi_2]\!]_G & := & \left\{(p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \mid\ (p_1, \mu_1) \in [\![\pi_1]\!]_G,\ (p_2, \mu_2) \in [\![\pi_2]\!]_G,\ \mu_1 \sim \mu_2, \text{ and } \mathrm{tgt}(p_1) = \mathrm{src}(p_2)\right\} \\
[\![\pi_1 + \pi_2]\!]_G & := & [\![\pi_1]\!]_G \cup [\![\pi_2]\!]_G \text{ only defined if } \mathsf{FV}(\pi_1) = \mathsf{FV}(\pi_2) \\
[\![\pi]\!]^0_G & := & \{(\mathrm{path}(n), \mu_\emptyset) \mid n \in N\} \\
[\![\pi]\!]^j_G & := & \left\{(p_1 \cdots p_j, \mu_\emptyset) \ \mid\ \exists \mu_1, \ldots, \mu_j :\ (p_1, \mu_1), \ldots, (p_j, \mu_j) \in [\![\pi]\!]_G \text{ and } \mathrm{tgt}(p_1) = \mathrm{src}(p_2), \ldots, \mathrm{tgt}(p_{j-1}) = \mathrm{src}(p_j)\right\} \\
[\![\pi^{i..m}]\!]_G & := & \bigcup_{j=i}^{m} [\![\pi]\!]^j_G \\
[\![\pi\langle\theta\rangle]\!]_G & := & \left\{(p, \mu) \in [\![\pi]\!]_G \ \mid\ \mu \models \theta\right\}
\end{array}
$$

$$
\begin{array}{rcl}
\mu \models x.k = y.k' & \Leftrightarrow & \rho(\mu(x), k) = \rho(\mu(y), k') \\
\mu \models x.k < y.k' & \Leftrightarrow & \rho(\mu(x), k) < \rho(\mu(y), k') \\
\mu \models \ell(x) & \Leftrightarrow & \lambda(\mu(x)) = \ell \\
\mu \models \theta \vee \theta' & \Leftrightarrow & \mu \models \theta \text{ or } \mu \models \theta' \\
\mu \models \theta \wedge \theta' & \Leftrightarrow & \mu \models \theta \text{ and } \mu \models \theta' \\
\mu \models \neg\theta & \Leftrightarrow & \mu \not\models \theta
\end{array}
$$

Figure 4: Semantics of patterns $[\![\pi]\!]_G$ and conditions $\mu \models \theta$ with respect to a property graph $G = (N, E, \mathrm{src}, \mathrm{tgt}, \lambda, \rho)$

for an untrained reader. The first attempt to produce a mathematical model of its pattern sublanguage (a 100-page part of the standard) was made in [50]. This model, called GPC (Graph Pattern Calculus), retained many key features of the standard; among them was a separation of variables into different kinds, leading to a complex type system that formed an integral part of GPC. A second attempt included not only pattern matching but all read-only operations of GQL, particularly operations that manipulate tables obtained by pattern matching [51].

Papers [50, 51] provided descriptions of (large parts of) GQL that can easily be grasped by academic researchers. Nevertheless, these abstractions are still complex to use a basic model for analyzing the expressive power of the languages. By an analogy with SQL, to understand its expressive power one needs to start with a simple abstraction like the relational calculus or algebra. In the spirit of searching for such a starting point for analyzing GQL, we now present a calculus from [56] that captures an "essential part" of GQL: a simple pattern calculus (without extras such as nulls or non-atomic values), and relational algebra as the language for manipulating pattern outputs. One important aspect that was treated [50, 51] and Section 3 but is not modeled here, however, is *list variables*.

4.1 CoreGQL

The language CoreGQL we present as an abstraction of GQL consists of three key components:

(1) a language for defining patterns;
(2) a way to turn patterns into relations; and
(3) a relational query language over such relations.

The easiest part of abstracting GQL into CoreGQL is (3): we just take relational algebra as such a language. As for (2), this is achieved by choosing a set of free variables of patterns, which in turn are designed in such a way that every relation we get from it is a first-normal-form relation (that is, it has no nulls, no duplicates, and every entry is an atomic value [28] — so, no path or list variables). We now look at all three components in more detail, starting with the main component of the language: its patterns. As we already announced in Remark 9, we omit backward edges, but they can easily be added.

4.1.1 CoreGQL Patterns. CoreGQL patterns are given by

$$
\begin{array}{rcll}
\pi & := & (x) & \text{node} \\
 & \mid & \overset{x}{\rightarrow} & \text{edge} \\
 & \mid & \pi_1\, \pi_2 & \text{concatenation} \\
 & \mid & \pi_1 + \pi_2 & \text{disjunction} \\
 & \mid & \pi^{n..m} & \text{repetition} \\
 & \mid & \pi\langle\theta\rangle & \text{condition}
\end{array}
$$

where variables x are optional, and $n \ldots m$ with $0 \le n \le m \le \infty$ indicates that a pattern is repeated between n and m times. Of course the Kleene star is simply $\pi^{0..\infty}$, so we shall write π^* for such patterns. Conditions are given by the grammar

$$\theta, \theta' \ := \ x.k = x'.k' \mid x.k < x'.k' \mid \ell(x) \mid \theta \vee \theta' \mid \theta \wedge \theta' \mid \neg\theta$$

where $x, x' \in \mathsf{Var}$, $k, k' \in \mathsf{Keys}$, and $\ell \in \mathsf{Labels}$.

Semantically, on a property graph, a pattern generates a set of pairs that consist of

- a path;
- a mapping of free variables to graph elements, i.e., nodes and edges.

To define this, we need the notion of *free variables* of patterns satisfying our key requirement that patterns generate first-normal-form relations, having no nulls and no non-atomic values. With this in mind, we define free variables as:

- $\mathsf{FV}(\ (x)\ ) = \mathsf{FV}\left(\overset{x}{\rightarrow}\right) := \{x\}$;
- $\mathsf{FV}(\pi_1\, \pi_2) := \mathsf{FV}(\pi_1) \cup \mathsf{FV}(\pi_2)$
- $\mathsf{FV}(\pi_1 + \pi_2) := \mathsf{FV}(\pi_1)$[2]

[2]We assume that in a pattern $\pi_1 + \pi_2$ it holds that $\mathsf{FV}(\pi_1) = \mathsf{FV}(\pi_2)$.

- $\mathsf{FV}(\pi^{n..m}) := \emptyset$
- $\mathsf{FV}(\pi\langle\theta\rangle) := \mathsf{FV}(\pi)$

Note that the penultimate rule ensures that there are no non-atomic values: a variable free in a pattern disappears from the list of free variables if the pattern gets repeated, ensuring that collections of bindings, viewed as relations, have only atomic values. The third rule ensures that there are no nulls.

Bindings are partial mappings from variables to graph elements, i.e., nodes and edges. Two bindings μ_1, μ_2 are *compatible*, written as $\mu_1 \sim \mu_2$, if they agree on their shared variables, i.e., if both μ_1 and μ_2 are defined on x then $\mu_1(x) = \mu_2(x)$. In this case, we define $\mu_1 \bowtie \mu_2$ by $(\mu_1 \bowtie \mu_2)(x) := \mu_1(x)$ if μ_1 is defined on x, and $(\mu_1 \bowtie \mu_2)(x) := \mu_2(x)$, otherwise. We let $\mu_\emptyset$ be the unique mapping with the empty domain. With this, the semantics of patterns and conditions is defined in Figure 4.

Before explaining how patterns produce relations, note one consequence of the semantics in Figure 4: every produced path is a node-to-node to path. Indeed, this is due to the way the semantics of base cases and composition is defined, and is consistent with the current state of languages such as Cypher and GQL.

4.1.2 Outputs of Patterns. Matching a pattern to a property graph $G = (N, E, \mathrm{src}, \mathrm{tgt}, \lambda, \rho)$ produces a relation. To specify this relation, let Ω be a sequence whose elements are variables x or expressions $x.k$ where x is a variable and k is a property. A mapping μ is *compatible* with Ω if for each $x \in \Omega$, μ is defined on x, and for each $x.k \in \Omega$, μ is defined on x and $\rho(\mu(x), k)$ is defined. We then define $\mu_\Omega : \Omega \to \mathsf{Nodes} \cup \mathsf{Edges} \cup \mathsf{Values}$ as

$$\mu_\Omega(\omega) := \begin{cases} \mu(x) & \text{if } \omega = x\,, \\ \rho(\mu(x), k) & \text{if } \omega = x.k\,. \end{cases}$$

Notice that the image of μ_Ω consists of atomic values, namely graph elements (nodes or edges) and property values.

With this, we define *patterns with output* as expressions of the form π_Ω whose semantics with respect to a property graph G is

$$[\![\pi_\Omega]\!]_G := \{\mu_\Omega \mid \mu \text{ is compatible with } \Omega \text{ and } \exists p : (p, \mu) \in [\![\pi]\!]_G\}$$

and the semantics of π is given in Figure 4. Since all mappings μ_Ω have the same domain Ω, we can view $[\![\pi_\Omega]\!]_G$ as a relation over the set of attributes Ω.

4.1.3 CoreGQL: Relational Operations Over Pattern Outputs. To define CoreGQL, we associate with each pattern π and a sequence Ω a relation symbol R^π_Ω of arity $|\Omega|$. The semantics of R^π_Ω is $[\![\pi_\Omega]\!]_G$, i.e., a first-normal-form relation over the set of attributes Ω. Then *CoreGQL* is defined as the set of relational algebra queries over all relations R^π_Ω.

As an example, consider a GQL query that returns nodes u and values of their property s such that u is connected to two different nodes u_1, u_2 with the same value of property p. For this, define patterns $\pi_i := (x) \to (x_i)$ and sequences $\Omega_i = (x, x.s, x_i, x_i.p)$ for $i = 1, 2$. Then the above query is

$$\pi_{x,x.s}\left(\sigma_{x_1 \neq x_2 \wedge x_1.p = x_2.p}\left(R^{\pi_1}_{\Omega_1} \bowtie R^{\pi_2}_{\Omega_2}\right)\right).$$

Note that here we use π both for (relational) projection and to denote a pattern (in π_1 and π_2).

4.2 CoreGQL vs GQL

To compare CoreGQL with actual GQL, note that the treatment of variables in the latter is more complicated. Specifically, a variable can be classified as one of the following types:

- a variable that binds a single graph element;
- a variable that binds a single graph element or null;
- a variable that binds a list of graph elements;
- a variable that binds a list of graph elements or null.

Which one of those categories a variable falls into depends on the context: it could be of one kind in a subpattern π' of π, but of a different kind in π itself. This is precisely the reason why in GQL $\pi^{2..2}$ is not the same as $\pi\pi$, as was already indicated in the introduction.

Our languages fixes this in different ways. The languages from Section 3.2 make it explicit when a variable is bound to a list, whereas CoreGQL simply disallows returning list-bound variables. This is done to avoid dealing with higher-order relations and to prove results on the expressiveness of GQL which we will see in the next section, but for now it means that the bottom-up and top-down approaches to abstracting graph query languages have not yet properly met in the middle to uncover and eliminate the monsters from Figure 1.

With respect to nulls, CoreGQL omits them too, thus forcing two sides of a disjunction $\pi_1 + \pi_2$ to have the same free variables. In actual GQL, this need not be the case, and *partial* mappings are allowed. For example, the pattern $\left((x) + \xrightarrow{y}\right)$ matches a node x or an edge y, producing bindings μ with domains $\{x\}$ or $\{y\}$. Interpreting μ as a tuple with attributes x, y requires marking one entry as a non-applicable null.

One significant difference between real GQL and all its theoretical reconstructions so far is that the former adopts *bag semantics* while the latter opt for avoiding duplicates. The perils of bag semantics in handling regular expressions being well known [9, 81], the design of GQL avoided complexity issues associated with them. Nevertheless, the interplay between deduplication and pattern matching in GQL leads to some counter-intuitive results, such as query results depending on whether a variable was given a name or not [35, Section 6].

Finally, GQL is a full-fledged query language with features such as aggregation, procedures, and updates, just to name a few; these have not been picked up by theoretical research so far.

5 Query Language Design: Where We Are

Building mathematical models of query languages and analyzing them is often the first step in producing recommendations for design adjustments and future enhancements. For example, the inexpressibility of recursive queries in extensions of first-order logic (FO) with aggregate functions [76] was the key motivation for adding recursive common table expressions to SQL, while the prohibitively high complexity of regular path queries in SPARQL led to changes in its standard [9, 81].

With graph languages, we are very early in this process: languages are new, and their formalizations are even newer, although the basic theory of RPQs and similar queries is fairly well established. Importantly, new languages such as SQL/PGQ and GQL are

in their first versions, and Cypher is not that old either; all of them will be undergoing changes and enhancements in the coming years. With this in mind, we can already offer an analysis of existing language features, not only identifying problematic elements of their design but also suggesting future corrections and improvements.

5.1 What is Missing?

Much of analysis of SQL expressiveness started with results on its expressive power, initially for simple theoretical languages like relational algebra and calculus. Now equipped with models of graph query languages, we present several results in this vein.

A Warm-Up: RPQs in Cypher. Unlike GQL and SQL/PGQ, Cypher puts limitations on patterns when it comes to repetitions: these can only be applied to edge labels or their disjunctions. For example, an RPQ ℓ^* can be straightforwardly expressed, but it seems that $(\ell\ell)^*$ cannot be. The claim that Cypher falls short of the full power of RPQs has been made several times [5, 33, 52, 107, 110]. It is however not easy to *prove* results about a real-life language without having a formal model of it (and in fact real-life Cypher gives one a backdoor to writing all RPQs, even though it is not very natural and is loaded with complexity issues, see Section 5.2).

To capture Cypher patterns for the purpose of this analysis, we adapt the definition of the pattern language from Section 4 as follows:

$$\pi \;:=\; (x:L) \;\mid\; \xrightarrow{x:L} \;\mid\; \xrightarrow{:L^*} \;\mid\; \pi_1\,\pi_2 \;\mid\; \pi_1 + \pi_2$$

where each L is a disjunction of labels $\ell_1 \mid \ell_2 \mid \cdots \mid \ell_n$. That is, arbitrary repetitions of patterns are disallowed, and only those applied to disjunctions of labels are kept. Since we are only concerned with RPQs and no data, we also removed conditions from the language, specifying labeling explicitly in node and edge patterns (such labels are assumed to include wildcard, making them effectively optional). With this definition, it is not hard to prove the following.

Proposition 22 ([55]). *The RPQ $(\ell\ell)^*$ is not expressible using Cypher patterns.*

Properties of Edges. For the next example of limitations of the expressive power of graph languages, we start with the following CoreGQL pattern:

$$\pi_{\mathsf{inc}} \;=\; (x)\;\big(\,((u)\to(v))\langle u.k < v.k\rangle\,\big)^*\;(y)$$

Free variables of π are x and y, and this pattern finds paths from x to y so that along them the value of property k of nodes increases.

But what if we wanted to ask for paths where property values of *edges* increase? A naive way of extending the previous query to capture the "increasing property values in edges" query does not work:

$$(x)\;\big(\,(\,()\xrightarrow{u}()\xrightarrow{v}()\,)\langle u.k < v.k\rangle\,\big)^*\;(y)$$

Indeed, this pattern will be matched on a four-edge path where values of property k of edges are $3, 4, 1, 2$ (in this order). This is because the subexpression $(\,()\xrightarrow{u}()\xrightarrow{v}()\,)\langle u.k < v.k\rangle$ moves forward in "steps of two" and the edge matching v in one iteration does not merge with the edge matching u in the next iteration, as happens for nodes in the pattern π_{inc}. The following shows that the increasing property values in edges cannot be expressed, under the assumption that the patterns do not use repeated variables.

Proposition 23 ([56]). *Patterns without repeated variables cannot express the "increasing property values in edges" query.*

We shall soon see what real-life Cypher and GQL do in order to amend this.

Reachability is Not Enough for NLOGSPACE. Already back in the 1980s we had graph languages that captured all NLogSpace queries [29]. Can CoreGQL do the same? Let us look at the following three statements:

- CoreGQL can express the reachability query: $(x) \to^{1..\infty} (y)$;
- Core GQL has at least the power of FO as it allows using relational algebra operators over the results of pattern matching;
- Reachability is complete for NLogSpace under first-order reductions.

Can we conlude that all NLogSpace queries are expressible in CoreGQL? No, we cannot.

Proposition 24 ([56]). *There are* DLogSpace *queries that are not expressible in CoreGQL.*

How is this possible? The problem is that to model first-order reductions, we need to test reachability in a structure obtained by applying a first-order transformation. However, in GQL, the flow of information is different: first patterns are evaluated (e.g., reachability), and only then are first-order operations used on pattern matching results. This flow of information in one direction only – from patterns to first-order operations – makes the language fall short of the full power of NLogSpace. What is missing is nesting (see Section 3.1.3). In fact, nesting is precisely what allows the language from [29] to capture NLogSpace.

5.2 Dangers of Ad-Hoc Solutions

Real-life languages are more expressive than their theoretical counterparts; SQL, for example, is Turing-complete as Turing machines can be simulated using recursion and aggregation. In response to users' demands for more expressiveness of graph languages, vendors include a number of solutions in their products. Some of these solutions, despite looking rather natural, lead to pitfalls. We discuss three of them, aiming to put back some issues – perhaps perceived as solved in industry – in the court of the language designer.

Matching on Matched Paths. There is a simple idea how to express queries such as "value in edges increases". For this, we must be able to match a new pattern *only* on the path that already matched some pattern. For this, we extend pattern matching with conditions of the form $\forall\pi' \Rightarrow \theta$. The meaning of $\pi\langle\forall\pi' \Rightarrow \theta\rangle$ is as follows. If p is a path that matches π, then π' is matched on p only, and every time it matches, the condition θ must be satisfied. Without giving the formal semantics of it, we illustrate the increasing value in edges query:

$$\big(\,(x)\;\to^*\;(y)\,\big)\langle\,\forall\big(\,\xrightarrow{u}()\xrightarrow{v}\,\big)\;\Rightarrow\;u.k < v.k\,\rangle$$

The match of $\xrightarrow{u}()\xrightarrow{v}$ happens on a matched path from x to y, and it is required that for every such match, i.e., for every two consecutive edges, the values of property k increase. While this seems a reasonable solution, and is advocated currently by the GQL committee [80, 116], there are unpleasant underwater currents.

Consider a slight variation

$$\big((x) \ \rightarrow^* (y) \big) \langle \forall \big((u) \rightarrow^* (v) \big) \ \Rightarrow \ u.k \neq v.k \rangle$$

This query looks for all paths from x to y on which all values of property k of nodes are different – and we know this query is NP-hard in data complexity [78].

Example 21 in Section 3.2.1 shows how to find paths with increasing values on edges using dl-RPQs. This, however, relies on the symmetric treatment of nodes and edges in dl-RPQs, which GQL currently does not have.

Turning to Complement for Help. A seemingly easy way to overcome the inexpressibility result shown above is to notice that its *complement* is expressible. Indeed, the *complement* of the condition that values in edges along a path increase states that there are consecutive edges in which property values do not increase. Since this latter condition is easily expressible in GQL, all that remains is to find all paths between two nodes, and from those subtract the set of paths that match the complement pattern.

Existing languages — Cypher, GQL, PGQ — all provide such facilities. They allow to name paths and allow path variables in outputs. That is, we can add patterns of the form $p = \pi$, where p is a path variable. Such a variable can become part of the output, resulting in a column in a relation whose entries are paths. So, for example, $(p = \pi)_p$ would return paths that are bound to the variable p. In this extension of the language, if we want to find all paths between nodes labeled ℓ_1 and ℓ_2 such that values in edges increase, we could write $\pi'_p - \pi''_p$ where

$$\begin{aligned} \pi' \quad &:= \quad p = \big((:\ell_1) \rightarrow^* (:\ell_2) \big) \\ \pi'' \quad &:= \quad p = \big((:\ell_1) \rightarrow^* () \xrightarrow{u} () \xrightarrow{v} () \rightarrow^* (:\ell_2) \big) \langle u.k \geq v.k \rangle \end{aligned}$$

(where the subscript p indicates that the output has paths themselves) since the pattern π'' expresses the negation of the condition "all values in edges increase" by checking that there is a pair of consecutive edges where the values do not increase.

So, while it is possible to express the "increasing values" property for edges, we see that we need completely different language ingredients than for expressing the same property for nodes. This puts an unnecessary burden on the user. Moreover, if we assume that the language is evaluated compositionally (first evaluate π', then π'', and then compute the difference), this might lead to poor performance, which is indeed observed in practice [56]. The underlying source of these problems is the asymmetric treatment of nodes and edges in the language design.

Turning to Lists for Help. We now turn to another way of overcoming expressivity limitations offered by practical systems. In Cypher, one can extract the list of nodes on a path bound to variable p using a function we write as $\mathtt{N}(p)$. Similarly, one can extract the list of edges using $\mathtt{E}(p)$. Over these lists one can perform the usual list operations. One of them is the `reduce` operation. It takes as a parameter a value ε, a unary function ι, and a binary function f. With that, $\mathtt{reduce}_{\varepsilon,\iota,f}(L)$ returns ε when the list L is empty, $\iota(x)$ when L has a unique element x, and $f(x, \mathtt{reduce}_{\varepsilon,\iota,f}(L'))$ when the head of L is x and its tail is L'.

As shown in [57], many queries become expressible with the help of list functions, including extensions of RPQs to different kinds of automata and even relations on words. We can also express the already discussed query that checks for paths with increasing values in edges. For simplicity, let us assume that we want the values to be non-negative. Let ι be the function that maps an edge e to the property value $e.k$, and let $f(e, v)$ be $e.k$ if $e.k \geq v \geq 0$ and -1 otherwise. Then we can find paths with increasing, non-negative values in edges using the following query

$$p \ = \ \big((x) \rightarrow^* (y) \big) \langle \mathtt{reduce}_{0,\iota,f}(\mathtt{E}(p)) \geq 0 \rangle \, .$$

This sounds promising, but the addition of lists and operations on them makes some infeasible queries deceptively easy to write. Consider for example the innocuously looking query

$$p \ = \ \big((x) \rightarrow^* (y) \big) \langle \mathtt{reduce}_{0,\iota,+}(\mathtt{E}(p)) = 0 \rangle$$

for the same ι as above. This query allows to encode the subset sum problem on a graph that is a sequence of nodes with parallel edges between each pair of consecutive nodes: one having the number, the other having zero. Therefore, this query is NP-complete in data complexity (even if matching paths p are restricted to be shortest, or simple, or trails). It can lead to evaluation issues even on tiny graphs with a few dozen nodes [57].

Lists can also be used to *aggregate* along paths. For example, the expression $\mathtt{reduce}_{0,\iota,+}(\mathtt{E}(p))$ seen earlier computes the sum of all values of property k of edges along path p. Let us denote it by Σ_p and consider the following

$$p = \big((:\ell) \rightarrow^+ (x:\ell) \big) \langle x.a \cdot (\Sigma_p)^2 + x.b \cdot \Sigma_p + x.c = 0 \rangle$$

assuming that the matching mode is shortest. There are two ways of providing a semantics to this query, depending on the point at which the condition is applied. To illustrate this, consider a very simple graph G, consisting of a single node u labeled ℓ with properties a, b, c, and a self-loop e on u with property k set to 1.

- If the condition is applied after shortest, then a single shortest path $\mathtt{path}(ueu)$ is computed, and the condition simply checks if $u.a + u.b + u.c = 0$.
- If shortest applies to paths that satisfy the condition, then a path is returned iff the equation $u.a \cdot x^2 + u.b \cdot x + u.c = 0$ has a positive integer solution; in fact the length of the path is precisely that solution.

The latter is uncomfortably close to solving Diophantine equations, and indeed with a slightly more complex (but still fixed) polynomial and a slightly bigger (but still bounded-size) graph, one can show that query answering is undecidable [50, 57]. This further shows that the list approach, while simple to use, comes loaded with problems that may not be easily seen by programmers.

6 Query Language Evaluation: Where We Are

Evaluating graph queries with list variables, path modes, and data filters efficiently is a challenging task that will require new techniques. In fact, if one does not design the query language carefully, evaluation can even become downright impossible.

6.1 Bag Semantics and Recursion: Boom!

Early versions of RPQs in SPARQL used bag semantics in combination with Kleene star [111], which easily result in prohibitively many answers [9]. Indeed, evaluating the RPQ $(((a^*)^*)^*)^*$ on a 6-clique with a-edges gave more answers than the number of protons

in the observable universe. The lesson: *if you put bag semantics together with recursion, the size of your outputs will explode.*

Where did this problem come from? Intuitively, the reason was that SPARQL's (multiset) union and join operators were re-used as union and concatenation in RPQs. If we define Kleene star using such multiset unions and joins, it means that the number of times that a pair of endpoints (x, y) should be returned by an RPQ R is the number of different ways that R can be matched on paths from x to y. This is, for a-labeled paths and expressions of the form $(((a^*)^*)^*)^*$, indeed *very many*, even if we restrict to simple paths.

Today, RPQs in SPARQL (called *property path expressions*) still have a strange (non-uniform) semantics: they use bag semantics for the union and concatenation operators, but set semantics for the Kleene star and Kleene plus operators [112]. In consequence, it is not clear which intuitive meaning we can associate to the number of times a pair of nodes is returned as an answer to an RPQ.

We believe that the underlying problem was twofold: *bag semantics* in combination with overly *syntax-driven design*. In this paper, we argue for a design of RPQs (and their incarnations with list variables and data tests) compatible with standard automata techniques (as in Sections 3.1.1,3.1.4, and 3.2.1), which allows avoiding both. For one thing, $(((a^*)^*)^*)^*$ can be equivalently rewritten to a^*. As we show next, automata techniques bring many other advantages.

6.2 Automata Techniques to the Rescue

From a principled point of view, the obvious approach to evaluating RPQs is to use automata. It was used by Mendelzon and Wood [89] and countless theory-oriented works afterwards (e.g., [25, 71, 78, 81, 87, 95]). Indeed, determining if a node pair (u, v) in graph G is an answer to an RPQ R is equivalent to testing if the NFA obtained from G by turning u into an initial state and v into an accepting state has a non-empty intersection with an NFA for R. The *product construction*, which is usually used for testing if the intersection is non-empty, can also be used to find multiple answers at once or constructed lazily if only one answer is required.

We assume familiarity with non-deterministic finite automata (NFAs) and write them as $(Q, \Sigma, \delta, q_0, F)$, where Q is the finite set of states, Σ is the finite alphabet of edge labels, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, q_0 is the initial state, and F is the set of accepting states. We recall that, given an RPQ R, an equivalent NFA N_R (without ε-transitions) can be constructed efficiently [100].

More precisely, given an edge-labeled graph $G = (N, E, \mathrm{src}, \mathrm{tgt}, \lambda)$ and NFA $N_R = (Q, \Sigma, \delta, q_0, F)$, the *product graph* $G_\times$ is then defined as the edge-labeled graph $G_\times = (N_\times, E_\times, \mathrm{src}_\times, \mathrm{tgt}_\times, \lambda_\times)$, where

- $V_\times = V \times Q$;
- $E_\times = \{(e, (q_1, a, q_2)) \in E \times \delta \mid \lambda(e) = a\}$;
- $\mathrm{src}_\times(e, (q_1, a, q_2)) = (\mathrm{src}(e), q_1)$
- $\mathrm{tgt}_\times(e, (q_1, a, q_2)) = (\mathrm{tgt}(e), q_2)$
- $\lambda_\times((e, (q_1, a, q_2))) = \lambda(e)$.

Each node of the form (u, q) in $G_\times$ corresponds to the node u in G and, furthermore, each path P of the form $(v, q_0), (v_1, q_1), \ldots, (v_n, q_n)$ in $G_\times$ corresponds to a path $p = v, v_1, \ldots, v_n$ in G that (a) has the same length as p and (b) brings the automaton from state q_0 to q_n. Consequently, when $q_n \in F$, this path in G matches R, meaning that $(v, v_n) \in [\![R]\!]_G$. In turn, this means that testing whether a pair (u, v) is an answer to an RPQ R over G boils down to solving a reachability problem from (u, q_0) to any (v, q) with $q \in F$. This gives a polynomial time algorithm both for determining whether a pair of nodes is a query answer, and for computing all such pairs. As we will see, this automata-based method can be extended for more expressive queries.

If we want to count the number of matching paths, it is important that N_R is *unambiguous*; that is, it has at most one accepting run per word. If this is the case, then the number of matching paths from u to v in G is the number of paths from (u, q_0) to any (v, q) with $q \in F$. A recent study of more than 150 million RPQs in SPARQL logs showed that, while ambiguous RPQs did occur, none of them required an unambiguous (or even deterministic) automaton that is larger than the regular expression [62].

The theme throughout this section will be to see where this simple idea can be extended to handle the features of modern graph query languages.

6.3 New Features Bring New Challenges

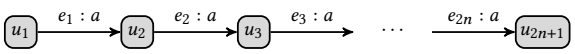
While we have a good understanding of how to evaluate RPQs using automata techniques [12], or how well these solutions work in practice [41, 110], the new features (a–d) from the introduction raise a host of new challenges that are yet to be addressed by the research community. While (a) —*handling both nodes and edges*— is perhaps mostly a language design challenge, (b–d) raise some significant challenges for query evaluation. We discuss these next.

Path and List Variables. Adding path and list variables to RPQs can make their outputs infinite. For example, evaluating the l-RPQ $(a^z)^*$, which matches a-labeled paths and, for each such path collects each edge in the list variable z, would result in an infinite set of paths when evaluated over any graph with an a-labeled cycle.

While CRPQs with list variables (Section 3.1.5) solve the infinity issue by allowing path restrictions (shortest, etc.), the number of results can still be prohibitively large. For instance, consider the following CRPQ with list variables which contains only one atom:[3]

$$q(z) :- \text{shortest } A(\mathrm{s}, \mathrm{t}) \ ,$$

with $A = (a^z)^*$, and consider the graph in Figure 5. The output of q on such graphs of size n consists of $2^{\Theta(n)}$ lists, due to the 2^n different paths that are matched. In fact, a list variable can even generate exponentially large output *on every path* that is matched by the query. This is illustrated by the path



and the l-RPQ $(aa^z + a^z a)^*$. Indeed, the list variable z will bind to 2^n different lists.

These prohibitively large query results reveal a major challenge in query processing: not only final query results cannot be directly presented to the user, but also intermediate query results cannot be materialised for further processing. This shows the need for (1) efficiently computable succinct representations of intermediate query results and (2) efficient query evaluation over such succinct representations further down the pipeline. Coming back to the principles of query language design, we note that these issues

[3]Here we generalize CRPQs by allowing atoms $R(t_1, t_2)$, where t_1 and t_2 are either variables in Var, or nodes in the graph. The semantics is based on homomorphisms that map each node to itself, while a variable can be mapped to any node.

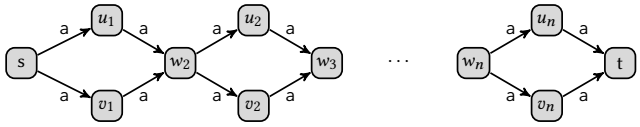


Figure 5: Graph with exponentially many paths from s to t.

already exist under set semantics. Under bag semantics, the outputs of queries become even larger.

Path Modes. In contrast to RPQs, which can be efficiently evaluated using the product construction (Section 6.2), deciding whether there is a *simple path* or *trail* from u to v that matches an RPQ in an edge-labeled graph G is NP-complete [13, 49, 74, 83, 89]. This means that many approaches for RPQ evaluation will not work anymore when only these restricted paths should be considered.

Data Filters. Data filters (Section 3.2.1) can significantly complicate query evaluation. For instance, they might require us to explore a large number of paths to detect one that satisfies all conditions in the query. To illustrate this, consider the property graph in Figure 3 and the query which asks for the shortest path of transfers from Mike to Rebecca, but such that at least one transfer with an amount less than 4.5M must be present. This query is easily expressible as an RPQ with data tests and shows that we need to search beyond shortest paths: the direct path between Mike and Rebecca, namely path(a3, t7, a5) is not a valid solution, but we need to use path(a3, t6, a4, t9, a6, t10, a5). The shortest mode in combination with data filters may even force using cycles: consider the shortest paths of transfers from Mike to Rebecca that has at least two transfers with an amount less than 4.5M. This example highlights the issue even when simple tests against constants are used; using the ability to store and later compare the values lead to even bigger challenges (we refer the reader to [78] for details).

6.4 What We Know

Next, we highlight what is known about some of these problems and where we see opportunities for future research.

Path Variables. Path variables were first studied in [15]. In general, it makes sense to use output-sensitive complexity measures to study (C)RPQ evaluation with path and list variables [34, 86], such as the framework of *enumeration algorithms* [101]. Since paths can grow arbitrarily long, and therefore *constant-delay* algorithms cannot exist, *output-linear delay* algorithms have been studied [41, 84].

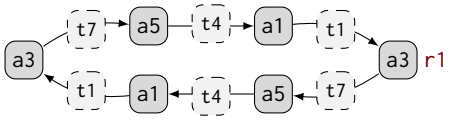
A fundamental step towards handling path and list variables is to be able to represent compactly the potentially huge number of intermediate results. A promising approach seems to be to represent (multi)sets of paths in *path multiset representations (PMRs)* [84]. PMRs are closely related to the product graph, but are more general — they do not necessarily involve an RPQ. Formally, a PMR over an edge labeled graph $G = (N_G, E_G, \mathrm{src}, \mathrm{tgt}, \lambda)$ is a tuple $R = (N, E, \mathrm{src}, \mathrm{tgt}, \gamma, S, T)$, where:

- $(N, E, \mathrm{src}, \mathrm{tgt})$ is an (unlabeled) graph;
- $\gamma : (N \cup E) \to (N_G \cup E_G)$ is a total homomorphism, meaning that it maps nodes in N to nodes in N_G, edges in E to edges in E_G, and for each $e \in E$ it holds that:
  - $\mathrm{src}(\gamma(e)) = \gamma(\mathrm{src}(e))$; and
  - $\mathrm{tgt}(\gamma(e)) = \gamma(\mathrm{tgt}(e))$.
- $S, T \subseteq N$ is a set of source and target nodes, respectively.

If R is a PMR over G, then each path $p = \mathrm{path}(n_1, \ldots, n_k)$ in R corresponds to a path $\gamma(p) = \mathrm{path}(\gamma(n_1), \ldots, \gamma(n_k))$ in G. Correspondingly, each PMR R over G represents the set of paths:

$$\mathrm{SPaths}(R) := \{\gamma(\rho) \mid \rho \text{ is a path from } S \text{ to } T \text{ in } R\}.$$

As their name suggests, PMRs support representing multisets of paths (see [84]), but we firmly believe that set semantics is a more natural fit for path queries, so we only consider this aspect of PMRs here. To illustrate how PMRs work, consider the property graph in Figure 3 and assume we wish to represent all (infinitely many) cycles of transfers from Mike to Mike which never pass through a blocked account. Effectively, these are the cycles defined by looping through the edges t7, t4 and t1. A PMR representing this information is depicted below, where $\gamma(o)$ is depicted inside of each object o, and with $S = T = \{\mathtt{r1}\}$:



Here we have an example of a finite representation of an infinite set of results, which is an interesting feature of PMRs. Similarly, if we consider again the graph from Figure 5, with an exponential number of paths we need to return. A PMR representing all these 2^n paths would essentially be the graph itself, with $S = \{\mathrm{s}\}$ and $T = \{\mathrm{t}\}$, taking space $O(n)$. Once constructed, a PMR can be passed to other operators processing a CRPQ with list variables without the need to explicitly enumerate paths one by one. Interestingly, the algorithms for returning paths in [41, 84] actually build a PMR representing the (set of) output paths in their pre-processing phase and use this PMR to enumerate the results efficiently. In this sense, PMRs are closely related to *factorized databases* [92]. However, PRMs represent results succinctly as finite state automata, whereas factorized databases use context-free grammars [72].

List Variables. To the best of our knowledge, the only work directly dealing with (C)RPQs with list variables [50, 51] only provides the formal definition of their semantics in GQL and SQL/PGQ. Our definition of l-CRPQs is designed to open up a connection to *capture variables* in *document spanners* [40]. Intuitively, document spanners are functions that extract mappings to substrings of a word. They are often defined using regular expressions with capture variables and their evaluation resembles how an RPQ with list variables operates on a single path [38, 47, 98]. For document spanners, we also need to handle the issue of exponentially many mappings over a single document, for which compact data structures have been introduced [2]. This allows efficient enumeration of output mappings in the fashion of enumeration algorithms mentioned before.

Path Modes. Recent work focused on single RPQs that only have path variables (or, equivalently, list variables that always match all graph elements on the path) [34, 41, 84, 86]. This is closely related to enumerating words in regular languages [1, 4] and there are specialized data structures developed to represent a large number of results [3, 90]. In its most systems-oriented incarnation [41],

this work leverages the product graph, but extends it so that paths of a specific type (shortest, trail, etc.) can be returned. Although determining whether a simple path or trail between two given nodes exists is already NP-complete [13, 83, 89], practical results show that this approach is actually feasible [41, 110] since queries and graphs used in practice are usually well behaved [62, 82, 87]. Works such as [34, 41, 110] also show how to avoid completing the entire pre-processing phase before staring to enumerate the results, which makes them highly relevant for real-world systems that implement a pipelined approach to query execution.

Data Filters. The idea of using variables to store values for data tests in graph-structured data (Section 3.2.1) comes from [78, 79]. The evaluation complexity for these expressions (without list variables) was studied only as a decision problem that checks whether a pair of nodes is connected by a path conforming to the RPQ with data tests, showing the problem to be PSPACE-complete in combined complexity and NLOGSPACE-complete in data complexity. These results use a variation of register automata [69] that operate on paths in a graph, and a modification of the product construction.

7 Where To Go: Road Map for Future Research

We conclude by outlining some directions for further study.

7.1 Language Design — Moderate Steps

It is not yet clear what exact role GQL will play in the development of graph languages. It could play a role of a pre-SQL language like QBE [117] or QUEL [104]. Or it could play the role of the first 1986 SQL standard that took a number of years to become what we know as SQL today. Either way, analyzing the expressive power and complexity of the current language design (and its abstractions) has a significant role to play in the development of future versions of GQL and SQL/PGQ.

Inexpressibility Toolkit. Query languages for property graphs are still fairly recent and their theoretical analysis has only just begun (Section 5 presented some early isolated results). The situation is somewhat similar to the state of finite model theory in the early days of relational languages. At that time, it produced isolated results, such as the inexpressibility of parity and transitive closure in first-order logic, and it took decades to develop a proper toolkit that allows us to use off-the-shelf tools, such as locality or zero-one laws, to prove more complex results [77]. Today, we are proving isolated results about particular graph queries and particular graph query languages, and the theory community has much to offer to help build a toolkit for analyzing graph languages at scale.

A Logic for Graphs. Theoretical analysis of relational query languages often relies on their connection to first-order logic and its extensions. A logic for graph query languages should give paths a central role. In standard relational queries, a single domain suffices. However, graph queries require logic that captures the structure of paths and their connection to nodes and edges. Standard many-sorted logic falls short because nodes, edges, and paths are not independent: Paths are formed from sequences of the two others. Hence, the logic should include constructs for navigating between these elements, for example, building a path from nodes and edges, retrieving path endpoints, etc. Two good starting points are the walk logic [65], designed for graph querying with support for path quantification, and the theory of concatenation [96], developed for strings but potentially adaptable to paths. Since the theory of concatenation is undecidable, we can consider its finite model counterpart which enjoys efficient model checking and captures various complexity classes when extended with operators for transitive closures or fixed point [54].

Evaluation Algorithms. In terms of query evaluation, the new features bring tons of challenges. Concerning *path and list variables*, the recurring story is that studies have looked at single RPQs, but little is known about CRPQs. For instance, it is interesting to study how compact representations for RPQ results interact with joins. But even for single RPQs, there are still interesting avenues to explore. We mentioned the framework of enumerating one output after another, but one could also study enumerating only the difference between consecutive outputs. Concerning paths, an interesting direction to look at could be Eppstein's data structure for enumerating the k shortest paths [39]. Concerning path modes, the current standards allow combinations, such as returning shortest concatenations of a trail and a simple path. To the best of our knowledge, the community has not even started investigating how to deal with such queries. Concerning data filters, an interesting next step is to see whether register automata can be extended to treat both nodes and edges symmetrically, as dl-RPQs do, and to see how to incorporate list variables into their runs. Of course, the main question here would be whether efficient enumeration algorithms could be designed, implemented, and integrated into query engines. Furthermore, we need to get a better idea of the size of intermediate query results in practice. Whereas existing practical studies focuses on structure of queries only [62, 82], we need to get a better idea of how these interact with the data.

Relational Algebra over Pattern Matching. Languages like GQL and SQL/PGQ apply relational algebra operators to relations extracted from graphs via pattern matching. There is a natural interplay between these two layers: some relational operations correspond to constructs in pattern matching, and can be pushed down to or lifted from the pattern matching layer. Exploring this interaction can support optimization, e.g., by reducing the size of intermediate results (similarly to techniques applied in the context of document spanners [37, 53]), and provide insights on the expressive power, e.g., by guiding the development of normal forms of queries. Another non-trivial question on the intersection with traditional techniques is how to develop cardinality estimation approaches for (C)RPQs. Finally, over the last decade we have seen impressive progress on worst-case optimal evaluation of conjunctive queries, with the celebrated AGM bound [11] and the subsequent race towards optimal algorithms. For CRPQs we have seen little progress so far, and some initial results show that it might be a challenging task [32, 70].

Parametrized Complexity. Aiming to mirror the successful line of research on conjunctive queries, spanning from the Yannakakis algorithm for evaluating acyclic CQs [115], through various alorithms for CQs with bounded treewidth and other width measures [26, 58, 60], and culminating in the celebrated dichotomies [27, 59, 88],

the academic community has been investigating parametrized complexity of CRPQs for over a decade now. Semantic treewidth, i.e., the minimal treewidth of an equivalent query, has been proposed as a candidate criterion to characterize fixed-parameter tractability of CRPQs [16, 99]. While equivalent queries with optimal treewidth can be computed and used for efficient query evaluation [42, 46], no dichotomies have been established so far.

Compositionality. CRPQs are not compositional, in the sense that they do not allow nesting, and neither are their extensions l-CRPQs and dl-CRPQs we have considered here. Meanwhile, SQL/PGQ and GQL allow using Kleene star over arbitrary patterns, which — together with the ability to use repeated variables for performing joins — gives them the full power of regular queries [50]. An important step in building faithful abstractions of graph query languages will be then to bring regular queries into the picture. A concrete challenge is reconciling regular queries with path modes.

Static Analysis. The complexity of query containment, the fundamental static analysis problem, is well understood for query languages working with edge-labeled graphs, such as CRPQs [23, 44, 45, 48] and regular queries [97]; for CRPQs there are even results on containment in the presence of schema constraints [61]. However, the effect of new features, such as list variables and data tests, is barely explored [73].

7.2 Language Design Revisited — Big Steps

The first versions of SQL/PGQ and GQL are already standardized [67, 68] and it is unclear if future versions will include major changes, such as treating nodes and edges symmetrically from the ground up or making the design of patterns fully compatible with automata. The latter would help make languages more declarative and amenable to optimization.

Theoretical research, however, does not need to be tied to compatibility with existing standards and can investigate freely how features such as (a)–(d) from the introduction can be added to query languages. In fact, theoretical guidance on these matters is extremely important to avoid ad-hoc solutions with unwanted side effects. Even if our community's results may not arrive in time for current versions of these languages, architectures come and go [102, 103] and query languages can be revised, but *theorems are forever* [108]. We believe that important principles to keep in mind when designing future (graph) query languages are (1) symmetry, (2) compatibility with automata, (3) set semantics, and (4) compositionality.

Finally, let us mention that, in our experience, input from the database theory community continues to be appreciated in query language design efforts. Some of us were involved in the standardization of SQL/PGQ and GQL since the beginning and, while not having full control, could steer the design towards more sustainable choices on several occasions. More recently, we were all involved in the design of Rel [8], a new language that aims at bridging the gap between query languages and programming in the large, and supports both relational and graph querying. The design of Rel takes wisdom from the database theory community seriously — notably, it uses set semantics — and we are excited to see how it will evolve in the future.

Acknowledgments

We would like to thank Molham Aref, with whom some of us had many discussions about language design. Molham's design principles, together with our research intuition that regular expressions should correspond to automata significantly helped to come up with our definition of (C)RPQs with data tests and list variables (Sections 3.2.1–3.2.2).

Libkin was supported by ANR project VeriGraph ANR-21-CE48-0015 and a direct grant from RelationalAI to the IRIF lab. Martens was supported by DFG grants MA 4938/2–1 and MA 4938/4–1. Murlak was supported by NCN grant 2018/30/E/ST6/00042. Peterfreund was supported by ISF grant 2355/24. Vrgoč was supported by ANID – Millennium Science Initiative Program – Code ICN17_002 and Fondecyt Regular project 1240346.

References

[1] Margareta Ackerman and Jeffrey O. Shallit. 2009. Efficient enumeration of words in regular languages. *Theor. Comput. Sci.* 410, 37 (2009), 3461–3470. doi:10.1016/J.TCS.2009.03.018

[2] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2021. Constant-Delay Enumeration for Nondeterministic Document Spanners. *ACM Trans. Database Syst.* 46, 1 (2021), 2:1–2:30. doi:10.1145/3436487

[3] Antoine Amarilli, Louis Jachiet, Martin Muñoz, and Cristian Riveros. 2022. Efficient Enumeration for Annotated Grammars. In *Symposium on Principles of Database Systems (PODS)*. ACM, 291–300. doi:10.1145/3517804.3526232

[4] Antoine Amarilli and Mikaël Monet. 2023. Enumerating Regular Languages with Bounded Delay. In *International Symposium on Theoretical Aspects of Computer Science (STACS) (LIPIcs, Vol. 254)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:18. doi:10.4230/LIPICS.STACS.2023.8

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

[6] Renzo Angles, Alin Deutsch, Thomas Frisendal, Victor Lee, Roi Lipman, Jeffrey Lovitz, Petra Selmer, Harsh Thakkar, Oskar van Rest, Mingxi Wu, and Boris Iordanov. 2019. GQL Influence Graph. gqlstandards.org/existing-languages.

[7] Renzo Angles, Aidan Hogan, Ora Lassila, Carlos Rojas, Daniel Schwabe, Pedro A. Szekely, and Domagoj Vrgoc. 2022. Multilayer graphs: a unified data model for graph databases. In *Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. ACM, 11:1–11:6. doi:10.1145/3534540.3534696

[8] Molham Aref, Paolo Guagliardo, George Kastrinis, Leonid Libkin, Victor Marsault, Wim Martens, Mary McGrath, Filip Murlak, Nathaniel Nystrom, Liat Peterfreund, Allison Rogers, Cristina Sirangelo, Domagoj Vrgoč, David Zhao, and Abdul Zreika. 2025. Rel: A Programming Language for Relational Data. In *International Conference on Management of Data (SIGMOD)*. doi:10.1145/3722212.3724450

[9] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting beyond a Yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *World Wide Web Conference (WWW)*. ACM, 629–638. doi:10.1145/2187836.2187922

[10] Marcelo Arenas and Jorge Pérez. 2011. Querying semantic web data with SPARQL. In *Symposium on Principles of Database Systems (PODS)*. ACM, 305–316. doi:10.1145/1989284.1989312

[11] Albert Atserias, Martin Grohe, and Dániel Marx. 2013. Size Bounds and Query Plans for Relational Joins. *SIAM J. Comput.* 42, 4 (2013), 1737–1767. doi:10.1137/110859440

[12] Pablo Barceló Baeza. 2013. Querying graph databases. In *Symposium on Principles of Database Systems (PODS)*. ACM, 175–188. doi:10.1145/2463664.2465216

[13] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2020. A trichotomy for regular simple path queries on graphs. *J. Comput. Syst. Sci.* 108 (2020), 29–48. doi:10.1016/J.JCSS.2019.08.006

[14] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. 2012. FDB: A Query Engine for Factorised Relational Databases. *Proc. VLDB Endow.* 5, 11 (2012), 1232–1243. doi:10.14778/2350229.2350242

[15] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4 (2012), 31:1–31:46. doi:10.1145/2389241.2389250

[16] Pablo Barceló, Miguel Romero, and Moshe Y. Vardi. 2016. Semantic Acyclicity on Graph Databases. *SIAM J. Comput.* 45, 4 (2016), 1339–1376.

[17] Henrik Björklund and Thomas Schwentick. 2010. On notions of regularity for data languages. *Theor. Comput. Sci.* 411, 4-5 (2010), 702–715. doi:10.1016/J.TCS.2009.10.009

[18] Mikolaj Bojanczyk and Slawomir Lasota. 2012. An extension of data automata that captures XPath. *Log. Methods Comput. Sci.* 8, 1 (2012). doi:10.2168/LMCS-8(1:5)2012

[19] Mikolaj Bojanczyk, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. 2009. Two-variable logic on data trees and XML reasoning. *J. ACM* 56, 3 (2009), 13:1–13:48. doi:10.1145/1516512.1516515

[20] Mikolaj Bojanczyk and Paweł Parys. 2011. XPath evaluation in linear time. *J. ACM* 58, 4 (2011), 17:1–17:33. doi:10.1145/1989727.1989731

[21] Pierre Bourhis, Markus Krötzsch, and Sebastian Rudolph. 2014. How to Best Nest Regular Path Queries. In *International Workshop on Description Logics (DL) (CEUR Workshop Proceedings, Vol. 1193)*. CEUR-WS.org, 404–415. https://ceur-ws.org/Vol-1193/paper_80.pdf

[22] Vicente Calisto, Benjamín Farias, Wim Martens, Carlos Rojas, and Domagoj Vrgoc. 2024. PathFinder Demo: Returning Paths in Graph Queries. In *ISWC 2024 Posters, Demos and Industry Tracks (CEUR Workshop Proceedings, Vol. 3828)*. CEUR-WS.org. https://ceur-ws.org/Vol-3828/paper34.pdf

[23] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2000. Containment of Conjunctive Regular Path Queries with Inverse. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. Morgan Kaufmann, 176–185.

[24] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2000. View-Based Query Processing for Regular Path Queries with Inverse. In *Symposium on Principles of Database Systems (PODS)*. ACM, 58–66. doi:10.1145/335168.335207

[25] Katrin Casel and Markus L. Schmid. 2021. Fine-Grained Complexity of Regular Path Queries. In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 186)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:20. doi:10.4230/LIPICS.ICDT.2021.19

[26] Chandra Chekuri and Anand Rajaraman. 2000. Conjunctive query containment revisited. *Theoretical Computer Science* 239, 2 (2000), 211–229.

[27] Hubie Chen, Georg Gottlob, Matthias Lanzinger, and Reinhard Pichler. 2020. Semantic Width and the Fixed-Parameter Tractability of Constraint Satisfaction Problems. In *International Joint Conference on Artificial Intelligence (IJCAI)*. 1726–1733.

[28] Edgar F. Codd. 1971. A Database Sublanguage Founded on the Relational Calculus. In *Proceedings of 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*. ACM, 35–68. doi:10.1145/1734714.1734718

[29] Mariano P. Consens and Alberto O. Mendelzon. 1990. GraphLog: a Visual Formalism for Real Life Recursion. In *Symposium on Principles of Database Systems (PODS)*. ACM Press, 404–416. doi:10.1145/298514.298591

[30] World Wide Web Consortium. 2014. RDF 1.1: Resource Description Framework (RDF) specification. W3C Recommendation. https://www.w3.org/TR/rdf11-concepts/

[31] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *International Conference on Management of Data (SIGMOD)*. 323–330. doi:10.1145/38713.38749

[32] Tamara Cucumides, Juan L. Reutter, and Domagoj Vrgoc. 2023. Size Bounds and Algorithms for Conjunctive Regular Path Queries. In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 255)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:17. doi:10.4230/LIPICS.ICDT.2023.13

[33] Claire David, Nadime Francis, and Victor Marsault. 2023. Run-Based Semantics for RPQs. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 178–187. doi:10.24963/KR.2023/18

[34] Claire David, Nadime Francis, and Victor Marsault. 2024. Distinct Shortest Walk Enumeration for RPQs. *Proc. ACM Manag. Data* 2, 2 (2024), 100. doi:10.1145/3651601

[35] Alin Deutsch, Nadime Francis, Alastair Green, Keith W. Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *International Conference on Management of Data (SIGMOD)*. ACM, 2246–2258. doi:10.1145/3514221.3526057

[36] Alin Deutsch and Val Tannen. 2001. Optimization Properties for Classes of Conjunctive Regular Path Queries. In *International Workshop on Database Programming Languages (DBPL)*. Springer, 21–39. doi:10.1007/3-540-46093-4_2

[37] Johannes Doleschal, Benny Kimelfeld, and Wim Martens. 2023. The Complexity of Aggregates over Extractions by Regular Expressions. *Log. Methods Comput. Sci.* 19, 3 (2023). doi:10.46298/LMCS-19(3:12)2023

[38] Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. 2019. Split-Correctness in Information Extraction. In *Symposium on Principles of Database Systems (PODS)*. ACM, 149–163. doi:10.1145/3294052.3319684

[39] David Eppstein. 1998. Finding the k Shortest Paths. *SIAM J. Comput.* 28, 2 (1998), 652–673. doi:10.1137/S0097539795290477

[40] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *J. ACM* 62, 2 (2015), 12:1–12:51. doi:10.1145/2699442

[41] Benjamín Farias, Wim Martens, Carlos Rojas, and Domagoj Vrgoc. 2024. PathFinder: Returning Paths in Graph Queries. In *International Semantic Web Conference (ISWC)*. 135–154. doi:10.1007/978-3-031-77850-6_8

[42] Cristina Feier, Tomasz Gogacz, and Filip Murlak. 2024. Evaluating Graph Queries Using Semantic Treewidth. In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 290)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:20. doi:10.4230/LIPICS.ICDT.2024.22

[43] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. 1997. A Query Language for a Web-Site Management System. *SIGMOD Rec.* 26, 3 (1997), 4–11. doi:10.1145/262762.262763

[44] Diego Figueira. 2020. Containment of UC2RPQ: The Hard and Easy Cases. In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 155)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:18.

[45] Diego Figueira, Adwait Godbole, S. Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. Containment of Simple Conjunctive Regular Path Queries. In *International Conference on Principles of Knowledge Representation and Reasoning (KR*. 371–380. doi:10.24963/KR.2020/38

[46] Diego Figueira and Rémi Morvan. 2023. Approximation and Semantic Treewidth of Conjunctive Regular Path Queries. In *International Conference on Database Theory (ICDT)*. https://hal.archives-ouvertes.fr/hal-03883042 Secondary link: https://www.morvan.xyz/papers/main-crpq-tw-icdt-v1.pdf.

[47] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2020. Efficient Enumeration Algorithms for Regular Document Spanners. *ACM Trans. Database Syst.* 45, 1 (2020), 3:1–3:42. doi:10.1145/3351451

[48] Daniela Florescu, Alon Y. Levy, and Dan Suciu. 1998. Query Containment for Conjunctive Queries with Regular Expressions. In *Symposium on Principles of Database Systems (PODS)*. ACM Press, 139–148.

[49] Steven Fortune, John Hopcroft, and James Wyllie. 1980. The directed subgraph homeomorphism problem. *Theoretical Computer Science (TCS)* 10, 2 (1980), 111–121.

[50] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. GPC: A Pattern Calculus for Property Graphs. In *Symposium on Principles of Database Systems (PODS)*. ACM, 241–250. doi:10.1145/3584372.3588662

[51] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoc. 2023. A Researcher's Digest of GQL (Invited Talk). In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 255)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:22. doi:10.4230/LIPICS.ICDT.2023.1

[52] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *International Conference on Management of Data (SIGMOD)*. ACM, 1433–1445. doi:10.1145/3183713.3190657

[53] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining Extractions of Regular Expressions. In *Symposium on Principles of Database Systems (PODS)*. ACM, 137–149. doi:10.1145/3196959.3196967

[54] Dominik D. Freydenberger and Liat Peterfreund. 2021. The Theory of Concatenation over Finite Models. In *International Colloquium on Automata, Languages, and Programming (ICALP) (LIPIcs, Vol. 198)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 130:1–130:17. doi:10.4230/LIPICS.ICALP.2021.130

[55] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. Database Theory in Action: Cypher, GQL, and Regular Path Queries. In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 328)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 36:1–36:5. doi:10.4230/LIPICS.ICDT.2025.36

[56] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. GQL and SQL/PGQ: Theoretical Models and Expressive Power. *PVLDB* 18, 6 (2025), 1798–1810.

[57] Amélie Gheerbrant, Leonid Libkin, and Alexandra Rogova. 2025. Dangers of List Processing in Querying Property Graphs. *Proc. ACM Manag. Data* 3, 3 (2025), 144:1–144:25.

[58] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree Decompositions and Tractable Queries. *J. Comput. System Sci.* 64, 3 (2002), 579–627.

[59] Martin Grohe. 2007. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM* 54, 1 (2007), 1:1–1:24.

[60] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms* 11, 1, Article 4 (aug 2014).

[61] Víctor Gutiérrez-Basulto, Albert Gutowski, Yazmín Angélica Ibáñez-García, and Filip Murlak. 2024. Containment of Graph Queries Modulo Schema. *Proc. ACM Manag. Data* 2, 2 (2024), 77. doi:10.1145/3651140

[62] Janik Hammerer and Wim Martens. 2025. A Compendium of Regular Expression Shapes in SPARQL Queries. (2025). Under submission.

[63] Olaf Hartig. 2017. Foundations of RDF★ and SPARQL★ (An Alternative Approach to Statement-Level Metadata in RDF). In *Alberto Mendelzon International Workshop on Foundations of Data Management and the Web (AMW) (CEUR Workshop Proceedings, Vol. 1912)*. CEUR-WS.org. http://ceur-ws.org/Vol-1912/paper12.pdf

[64] Olaf Hartig, Pierre-Antoine Champin, Gregg Kellogg, Andy Seaborne, Dörthe Arndt, Jeen Broekstra, Bob DuCharme, Ora Lassila, Peter F. Patel-Schneider, Eric Prud'hommeaux, Ted Thibodeau Jr., and Bryan Thompson. 2021. RDF-star and SPARQL-star. W3C Draft Community Group Report. https://w3c.github.io/rdf-star/cg-spec/2021-07-01.html

[65] Jelle Hellings, Bart Kuijpers, Jan Van den Bussche, and Xiaowang Zhang. 2013. Walk logic as a framework for path query languages on graph databases. In *International Conference on Database Theory (ICDT)*. ACM, 117–128. doi:10.1145/2448496.2448512

[66] Filip Ilievski, Daniel Garijo, Hans Chalupsky, Naren Teja Divvala, Yixiang Yao, Craig Milo Rogers, Ronpeng Li, Jun Liu, Amandeep Singh, Daniel Schwabe, and Pedro A. Szekely. 2020. KGTK: A Toolkit for Large Knowledge Graph Manipulation and Analysis. In *International Semantic Web Conference (ISWC)*. Springer, 278–293.

[67] ISO/IEC JTC 1/SC 32. 2023. *ISO/IEC 9075-16:2023. Information technology – Database languages SQL. Part 16: Property Graph Queries (SQL/PGQ)*. Technical Report. ISO.

[68] ISO/IEC JTC 1/SC 32. 2024. *ISO/IEC 39075:2024. Information technology – Database languages GQL*. Technical Report. ISO.

[69] Michael Kaminski and Nissim Francez. 1994. Finite-memory automata. *Theoretical Computer Science* 134, 2 (1994), 329–363.

[70] Nikolaos Karalis, Alexander Bigerl, Liss Heidrich, Mohamed Ahmed Sherif, and Axel-Cyrille Ngonga Ngomo. 2024. Efficient Evaluation of Conjunctive Regular Path Queries Using Multi-way Joins. In *International Conference on The Semantic Web (ESWC)*. Springer, 218–235. doi:10.1007/978-3-031-60626-7_12

[71] Mahmoud Abo Khamis, Ahmet Kara, Dan Olteanu, and Dan Suciu. 2024. Output-Sensitive Evaluation of Regular Path Queries. *CoRR* abs/2412.07729 (2024). doi:10.48550/ARXIV.2412.07729 arXiv:2412.07729

[72] Benny Kimelfeld, Wim Martens, and Matthias Niewerth. 2025. A Formal Language Perspective on Factorized Representations. In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 328)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 20:1–20:20. doi:10.4230/LIPICS.ICDT.2025.20

[73] Egor V. Kostylev, Juan L. Reutter, and Domagoj Vrgoc. 2014. Containment of Data Graph Queries. In *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy (Eds.). OpenProceedings.org, 131–142. doi:10.5441/002/ICDT.2014.16

[74] Andrea S. LaPaugh and Christos H. Papadimitriou. 1984. The even-path problem for graphs and digraphs. *Networks* 14, 4 (1984), 507–513.

[75] Ora Lassila, Michael Schmidt, Olaf Hartig, Brad Bebee, Dave Bechberger, Willem Broekema, Ankesh Khandelwal, Kelvin Lawrence, Carlos Manuel López Enríquez, Ronak Sharda, and Bryan B. Thompson. 2023. The OneGraph vision: Challenges of breaking the graph model lock-in. *Semantic Web* 14, 1 (2023), 125–134. doi:10.3233/SW-223273

[76] Leonid Libkin. 2004. *Elements of Finite Model Theory*. Springer. doi:10.1007/978-3-662-07003-1

[77] Leonid Libkin. 2009. The finite model theory toolbox of a database theoretician. In *Symposium on Principles of Database Systems (PODS)*. ACM, 65–76. doi:10.1145/1559795.1559807

[78] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. 2016. Querying Graphs with Data. *J. ACM* 63, 2 (2016), 14:1–14:53. doi:10.1145/2850413

[79] Leonid Libkin and Domagoj Vrgoc. 2012. Regular path queries on graphs with data. In *International Conference on Database Theory (ICDT)*. ACM, 74–85. doi:10.1145/2274576.2274585

[80] Tobias Lindaaker. 2023. *Predicates on sequences of edges*. Technical Report. ISO/IEC JTC1/ SC32 WG3:W26-027.

[81] Katja Losemann and Wim Martens. 2013. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.* 38, 4 (2013), 24. doi:10.1145/2494529

[82] Wim Martens. 2022. Towards Theory for Real-World Data. In *Symposium on Principles of Database Systems (PODS)*. ACM, 261–276. doi:10.1145/3517804.3526066

[83] Wim Martens, Matthias Niewerth, and Tina Popp. 2023. A Trichotomy for Regular Trail Queries. *Log. Methods Comput. Sci.* 19, 4 (2023). doi:10.46298/LMCS-19(4:20)2023

[84] Wim Martens, Matthias Niewerth, Tina Popp, Carlos Rojas, Stijn Vansummeren, and Domagoj Vrgoc. 2023. Representing Paths in Graph Database Pattern Matching. *Proc. VLDB Endow.* 16, 7 (2023), 1790–1803. doi:10.14778/3587136.3587151

[85] Wim Martens and Tina Popp. 2022. The Complexity of Regular Trail and Simple Path Queries on Undirected Graphs. In *Symposium on Principles of Database Systems (PODS)*. ACM, 165–174. doi:10.1145/3517804.3524149

[86] Wim Martens and Tina Trautner. 2018. Evaluation and Enumeration Problems for Regular Path Queries. In *International Conference on Database Theory (ICDT) (LIPIcs, Vol. 98)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 19:1–19:21. doi:10.4230/LIPICS.ICDT.2018.19

[87] Wim Martens and Tina Trautner. 2019. Dichotomies for Evaluating Simple Regular Path Queries. *ACM Trans. Database Syst.* 44, 4 (2019), 16:1–16:46. doi:10.1145/3331446

[88] Dániel Marx. 2010. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. In *Symposium on Theory of Computing (STOC)*. 735–744.

[89] Alberto O. Mendelzon and Peter T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *Very Large Data Bases*. 185–193. http://www.vldb.org/conf/1989/P185.PDF

[90] Martin Muñoz and Cristian Riveros. 2024. Streaming Enumeration on Nested Documents. *ACM Trans. Database Syst.* 49, 4 (2024), 15:1–15:39. doi:10.1145/3701557

[91] Neo4j. 2025. Cypher Manual. https://neo4j.com/docs/cypher-manual/.

[92] Dan Olteanu and Jakub Zavodny. 2012. Factorised representations of query results: size bounds and readability. In *International Conference on Database Theory (ICDT)*. ACM, 285–298. doi:10.1145/2274576.2274607

[93] Oracle. [n. d.]. Oracle Graph Database. https://www.oracle.com/database/graph/.

[94] Jorge Pérez, Marcelo Arenas, and Claudio Gutiérrez. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3 (2009), 16:1–16:45.

[95] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2010. nSPARQL: A navigational language for RDF. *J. Web Semant.* 8, 4 (2010), 255–270. doi:10.1016/J.WEBSEM.2010.01.002

[96] Willard V Quine. 1946. Concatenation as a basis for arithmetic. *The Journal of Symbolic Logic* 11, 4 (1946), 105–114.

[97] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2017. Regular Queries on Graph Databases. *Theory Comput. Syst.* 61, 1 (2017), 31–83. doi:10.1007/S00224-016-9676-2

[98] Cristian Riveros, Nicolás Van Sint Jan, and Domagoj Vrgoc. 2023. REmatch: a novel regex engine for finding all matches. *Proc. VLDB Endow.* 16, 11 (2023), 2792–2804. doi:10.14778/3611479.3611488

[99] Miguel Romero, Pablo Barceló, and Moshe Y. Vardi. 2017. The homomorphism problem for regular graph patterns. In *Symposium on Logic in Computer Science (LICS)*. 1–12.

[100] Jacques Sakarovitch. 2009. *Elements of automata theory*. Cambridge University Press.

[101] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *International Conference on Database Theory (ICDT)*. ACM, 10–20.

[102] Michael Stonbraker and Joe Hellerstein. 2005. *Readings in Database Systems* (4 ed.). Chapter What Goes Around Comes Around, 2–41.

[103] Michael Stonebraker and Andrew Pavlo. 2024. What Goes Around Comes Around... And Around.. *SIGMOD Rec.* 53, 2 (2024), 21–37. doi:10.1145/3685980.3685984

[104] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. 1976. The Design and Implementation of INGRES. *ACM Trans. Database Syst.* 1, 3 (1976), 189–222. doi:10.1145/320473.320476

[105] Memgraph Team. 2023. Memgraph. https://memgraph.com/

[106] TigerGraph Team. 2021. TigerGraph Documentation – version 3.1. https://docs.tigergraph.com/

[107] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Fourth International Workshop on Graph Data Management Experiences and Systems*. 1–6.

[108] Moshe Vardi. 2006. Personal motto. (2006). Quote published in [113].

[109] Vesoft Inc/Nebula. 2023. NebulaGraph. https://www.nebula-graph.io/

[110] Domagoj Vrgoc, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil-Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2023. MillenniumDB: An Open-Source Graph Database System. *Data Intell.* 5, 3 (2023), 560–610. doi:10.1162/DINT_A_00229

[111] W3C Sparql 2012. SPARQL 1.1 Query Language. https://www.w3.org/TR/2012/WD-sparql11-query-20120105/. World Wide Web Consortium.

[112] W3C Sparql 2013. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/. World Wide Web Consortium.

[113] Marianne Winslett. 2006. Moshe Vardi speaks out on the proof, the whole proof, and nothing but the proof. *SIGMOD Rec.* 35, 1 (2006), 56–64. doi:10.1145/1121995.1122008

[114] Peter T. Wood. 2012. Query languages for graph databases. *SIGMOD Rec.* 41, 1 (2012), 50–60. doi:10.1145/2206869.2206879

[115] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*. 82–94.

[116] Fred Zemke. 2024. *For each segment discussion*. Technical Report. ISO/IEC JTC1/SC32 WG3:BGI-022.

[117] Moshé M. Zloof. 1977. Query-by-Example: A Data Base Language. *IBM Syst. J.* 16, 4 (1977), 324–343. doi:10.1147/SJ.164.0324