GQL and SQL/PGQ: Theoretical Models and Expressive Power

Amélie Gheerbrant
Université Paris Cité, CNRS, IRIF
Paris, France
amelie@irif.fr

Leonid Libkin
RelationalAI & University of Edinburgh
Paris & Edinburgh, UK & France
l@libk.in

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

Alexandra Rogova
Université Paris Cité, CNRS, IRIF
Paris, France
rogova@irif.fr

ABSTRACT

SQL/PGQ and GQL are very recent international standards for querying property graphs: SQL/PGQ specifies how to query relational representations of property graphs in SQL, while GQL is a standalone language for graph databases. The rapid industrial development of these standards left the academic community trailing in its wake. While digests of the languages have appeared, we do not yet have concise foundational models like relational algebra and calculus for relational databases that enable the formal study of languages, including their expressiveness and limitations. At the same time, work on the next versions of the standards has already begun, to address the perceived limitations of their first versions.

Motivated by this, we initiate a formal study of SQL/PGQ and GQL, concentrating on their concise formal model and expressiveness. For the former, we define simple core languages – Core PGQ and Core GQL – that capture the essence of the new standards, are amenable to theoretical analysis, and clarify the difference between PGQ's bottom up evaluation versus GQL's linear, or pipelined approach. Equipped with these models, we both confirm the necessity to extend the language to fill in the expressiveness gaps and identify the source of these deficiencies. We complement our theoretical analysis with an experimental study, demonstrating that existing workarounds in full GQL and PGQ are impractical, further underscoring the necessity to correct deficiencies in language design.

KEYWORDS

Graph databases, GQL, SQL/PGQ, Cypher, pattern matching, expressive power, language design

PVLDB Reference Format:
Amelie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. GQL and SQL/PGQ: Theoretical Models and Expressive Power. PVLDB, 18(6): 1798 - 1810, 2025.
doi:10.14778/3725688.3725707

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.com/alexandra-rogova/neo4j_increasing_value_test.

Proceedings of the VLDB Endowment, Vol. 18, No. 6 ISSN 2150-8097.
doi:10.14778/3725688.3725707

1 INTRODUCTION

In the past two years, ISO published two new international standards. SQL/PGQ, released in 2023, as Part 16 of the SQL standard, provides a mechanism for representing and querying property graphs in relational databases using SQL. GQL, released in 2024, is a standalone graph query language that does not rely on a relational representation of graphs. SQL/PGQ and GQL are developed in the same ISO committee that has been maintaining and enhancing the SQL standard for decades. These developments reflect the interest of relational vendors in implementing graph extensions and the emergence of a new native graph database industry.

There is however a notable difference between this standardization effort and that of SQL as it happened in the 1980s. Before SQL was designed, strong theoretical and practical foundations of relational databases had been developed. Equivalence of relational algebra and calculus (first-order logic) had been known; these provided very clean abstractions of relational languages that served as the basis of relational database theory. This has had a profound impact on both the theory and the practice of database systems. For example, a question of central importance in the early days of relational databases was expressiveness of query languages, with Fagin, Aho, Ullman, Gaifman, and Vardi independently showing that relational calculus cannot define the transitive closure of a relation [4, 18, 23]. This led to subsequent development of Datalog and culminated in the inclusion of linear recursive queries by the SQL standard in 1999 [17].

When it comes to graphs, the adoption of languages by industry runs well ahead of the development of their foundational underpinnings. For many years, work on graph queries concentrated on regular path queries (RPQs) [14] and their multiple extensions (e.g., [5, 7, 8, 12, 13]). These however assume a much simplified model in which neither nodes nor edges have properties, unlike *property graphs* preferred in industry. A step in that direction is the model of *data graphs* in which nodes but not edges carry data [30]. To this day, these models are the basis of work on query language expressiveness [10, 32]. But we will see soon that from the point of view of GQL and SQL/PGQ, a full property graph model makes a huge difference. Existing theoretical languages such as GXPath [30], regular queries [34], STRUQL [19], while having a direct influence on the design of GQL [35], are not a good reflection of it.

At the same time, extensive discussions in the standards committees are already under way to identify new features of SQL/PGQ and GQL, based on their *perceived*, rather than proved, shortcomings [31, 43]. These considerations motivate our main contributions:

(1) We provide concise formal models of SQL/PGQ and GQL;
(2) We confirm the intuition that certain queries of interest to customers of graph databases cannot be expressed by SQL/PGQ and GQL patterns;
(3) We compare GQL with recursive SQL and show that the latter is more powerful; and
(4) We show experimentally that methods currently allowed in SQL/PGQ and GQL to overcome these limitations are impractical, even for small-sized graphs.

We now elaborate more on these goals.

Formal models of GQL and SQL/PGQ. The workhorse of graph query languages is *pattern matching*. In Cypher, PGQ, GQL and others, pattern matching turns a graph into a table; the remaining operations of the language manipulate that table. In fact in GQL and SQL/PGQ pattern matching is *identical*; it is only how its results are processed that is different. In SQL/PGQ, a graph is a view defined on a relational database. The result of pattern matching is simply a table in the `FROM` clause of a SQL query that may use other relations in the database. GQL, on the other hand, is oblivious to how a graph is stored, and the table resulting from pattern matching is manipulated by a sequence of operators that modify it, in an imperative style that is referred to as "linear composition".

To produce a formal model of these languages, we use the relationship between SQL and relational calculus/algebra as the guiding principle. SQL has a multitude of features: bag semantics, nulls, arithmetic operations, aggregate functions, outerjoins, complex datatypes such as arrays. Some of these break the theoretical model of First Normal Form (1NF) relations which are *sets* of tuples of *atomic values*. Relational algebra and calculus get rid of the extra baggage that SQL is forced to add to be usable in practice, and yet provide a simple core over sets of tuples of atomic values.

Although some mathematical abstractions of GQL and SQL/PGQ exist [20, 21], they are not yet at the same level as RA or relational calculus, in terms of their simplicity and utility in formally proving results. For example, their formalizations of pattern matching come with a complex typing system and the use of conditional and group variables that create nulls and set-valued attributes. To achieve our goal of creating a simple and usable abstraction, we need to: simplify the model of pattern matching, to avoid outputting non-1NF relations, and to formalize the linear composition of Cypher and GQL. This allows us to define two languages – Core PGQ and Core GQL – as RA or linear composition of relational operators on top of pattern matching outputs.

Limitations of pattern matching. Many pattern matching tasks require iterating patterns, notably when we do not know a priori the length of a path we are searching for (a basic example is reachability in graphs). The way pattern matching is designed in Cypher, GQL, and SQL/PGQ, makes it easy to compare two consecutive iterations of a pattern based on property values of their nodes, but at the same time it is very hard to compare property values in *edges*.

To illustrate this, assume we have a chain of transfers between accounts: accounts are nodes, with property values such as "balance", and transfers are edges, with property values such as "amount" and "timestamp", between accounts. It is very easy to write a query looking for a chain of transfers such that the *balance* increases in accounts along the chain. It appears to be very hard or impossible to write a query looking for a chain of transfers such that the *timestamp* increases along the path.

Demand for these queries occurs often in practice, which forced several companies participating in GQL design in ISO to start thinking of language extensions that will permit such queries [31, 43]. However, before making non-trivial language enhancements, it would be good to actually know that newly added features cannot be achieved with what is already in the language.

We show that this is indeed the case. Equipped with our formalization of the pattern language, we prove that it cannot express a large class of queries that analyze how property values of edges change along matched paths; of these the *increasing value in edges* query is the simplest. This follows in fact from a more general property that is akin to a pumping lemma for paths that can be selected by GQL and PGQ patterns.

GQL vs Recursive SQL vs Datalog. When the ubiquitous CRPQs was introduced in [13], it was shown that a Datalog-like language based on CRPQs has the power of transitive closure logic [29] and captures the complexity class NLogSpace of problems solved by nondeterministic Turing machines whose work tape used logarithmic number of bits in terms of the size of the input (this class is contained in polynomial time). Since then, NLogSpace is the yardstick complexity class we compare graph languages to, and Datalog – that subsumes the transitive closure logic – is a typical language used to understand the power of graph querying. Datalog is also the basis of SQL's recursive common table expressions (CTE) introduced by the `WITH RECURSIVE` clause. In fact, in SQL only *linear Datalog* rules are allowed: in them, the recursively defined predicate can be used at most once.

Motivated by this, we compare the power of GQL with recursive SQL and linear Datalog. We show that there are queries that are expressible in the latter, have very low data complexity, and yet are *not expressible* in GQL. We explain how this points out deficiencies of GQL design that will hopefully be addressed in the future.

Experimental evaluation: can graph DBMSs overcome these limitations? Real-life systems go beyond basic calculi; SQL with recursion and aggregates is in fact Turing-complete and thus can express all computable queries. Similarly, GQL, PGQ, Cypher and others have many tools at their disposal to let users write very powerful queries. In fact queries such as *increasing value in edges* are expressible in real-life Cypher and PGQ, though in a very convoluted way. Since the *complement* of the query is easily expressible, one can look for all the paths and then subtract the complement. Intuitively, such a way should not work in practice as the number of paths in a graph grows exponentially.

This is precisely what we confirm using three different implementation of graph database queries: Neo4j and Memgraph for Cypher, and DuckDB for SQL/PGQ. Native graph systems can handle just a few dozen nodes before 100% timeout rate is observed; DuckDB does marginally better as it computes fewer paths to start with. It is important to note that this is not a critique of the systems tested, as the indirect method of computing the query inherently requires an exponential number of paths. Instead, this confirms that no workaround can bypass the inherent expressibility limitations.

Since inexpressibility results are commonly used to identify deficiencies in language design, and serve as a motivation to increase language expressiveness, we shall at the end of the paper discuss what these results tell us in terms of new features of GQL and SQL/PGQ that will be required.

Related work. While industry is dominated by property graphs (Neo4j [22], Oracle [40], Amazon [9], TigerGraph [16], etc), much of academic literature still works with the model of labeled graphs and query languages based on RPQs, with some exceptions [2, 30, 36]. However, the rare models focused on property graphs appeared before the new standards became available, and their analyses of expressiveness and language features do not apply to GQL and SQL/PGQ. The first commercial language for property graphs was Cypher, and it was fully formalized in [22]. As GQL and SQL/PGQ were being developed, a few academic papers appeared. For example, [15] gave an overview of their pattern matching facilities, which was then further analyzed in [20], and [39] provided a description of an early implementation of SQL/PGQ.

In [21], a digest of GQL suitable for the research community was presented. While a huge improvement compared to the actual standard from the point of view of clarity, the presentation of [21] replaced 500 pages of the text of the Standard (notoriously hard to read) by a one-page long definition of the syntax, followed by a four-pages long definition of the semantics. It achieved a two orders of magnitude reduction in the size of the definition of the language, but 5 pages is still way too long for "Definition 1". The language of [20] is closer to our goal, but it is not well-suited to the level reasoning we require here as it still contains still too much of the GQL and PGQ baggage, for example non-1NF relations, and a complex type system for singleton, conditional, and group variables that we avoid here entirely and replace by the very familiar definition of free variables.

Another element missing from the literature is a proper investigation of linear composition. Initially introduced in Cypher, it was then adopted in a purely relational language PRQL [33] (positioned as "pipelined" alternative to SQL), embraced by GQL, and influenced the designed of the piped SQL syntax [37]; however formal analyses of this way of building complex queries are lacking.

2 GQL AND SQL/PGQ BY EXAMPLES

We illustrate GQL and SQL/PGQ capabilities using the graph from Figure 1 and the following money-laundering query: *Find a pair of friends in the same city who transfer money to each other via a common friend who lives elsewhere.* Notice that we assume that friendship is a symmetric relation.

In GQL, it will be expressed as query:

```
MATCH (x)-[:Friends]->(y)-[:Friends]->(z)
-[:Friends]->(x), (x)-[:Owns]->(acc_x),
(y)-[:Owns]->(acc_y), (z)-[:Owns]->(acc_z),
(acc_x)-[t1:Transfer]->(acc_z)
-[t2:Transfer]->(acc_y)
FILTER (y.city = x.city) AND (x.city<>z.city)
AND (t2.amount < t1.amount)
RETURN x.name AS name1, y.name AS name2
```

In SQL/PGQ, a graph is a view of a tabular schema. The graph from Figure 1 can be represented by the following set of tables:

- `Person(p_id,name,city)` for people,
- `Acc(a_id,type)` for accounts,
- `Friend(p_id1,p_id2,since)` for friendships,
- `Owns(a_id,p_id)` for ownership and
- `Transfer(t_id,a_from,a_to,amount)` for transfers.

The property graph view is then defined by a `CREATE` statement, part of which is shown below:

```
CREATE PROPERTY GRAPH Interpol1625 (
VERTEX TABLES
Acc KEY (a_id) LABEL Account PROPERTIES (type)
....
EDGE TABLES
Transfer KEY (t_id)
SOURCE KEY (a_from) REFERENCES Acc
DESTINATION KEY (a_to) REFERENCES Acc
LABEL Transfer PROPERTIES (amount)
.... )
```

This view defines nodes (vertices) and edges of the graph, specifies endpoints of edges, and defines their labels and properties. We can then query it, using pattern matching to create a subquery:

```
SELECT T.name_x AS name1, T.name_y AS name2
FROM Interpol1625 GRAPH_TABLE (
MATCH (x)-[:Friends]->(y)-[:Friends]->(z)
-[:Friends]->(x),  (x)-[:Owns]->(acc_x),
(y)-[:Owns]->(acc_y), (z)-[:Owns]->(acc_z),
(acc_x)-[t1:Transfer]->(acc_z)
-[t2:Transfer]->(acc_y)
COLUMNS x.city AS city_x, y.city AS city_y,
z.city AS city_z,
x.name AS name_x, y.name AS name_y,
t1.amount AS amnt1, t2.amount AS amnt2
) AS T
WHERE T.city_x=T.city_y
AND T.city_x <> T.city_z
AND T.amount1 > T.amount2
```

Note that in GQL, a sequence of operators can continue *after* the `RETURN` clause. For example, if we want to find large transfers between the two potential offenders we could simply continue the first GQL query with extra clauses:

```
MATCH (u WHERE u.name=name1)
-[t:Transfer]->
(v WHERE v.name=v2)
FILTER t.amount > 100000
RETURN t.amount AS big_amount
```

This is what is referred to as *linear composition*: we can simply add clauses to the already existing query which apply new operations to the result of already processed clauses.

In SQL/PGQ, such an operation is also possible, though perhaps a bit more cumbersome as we would need to put the above PGQ query as a subquery in `FROM` and create another subquery for the second match, then join them on `name1` and `name2`.

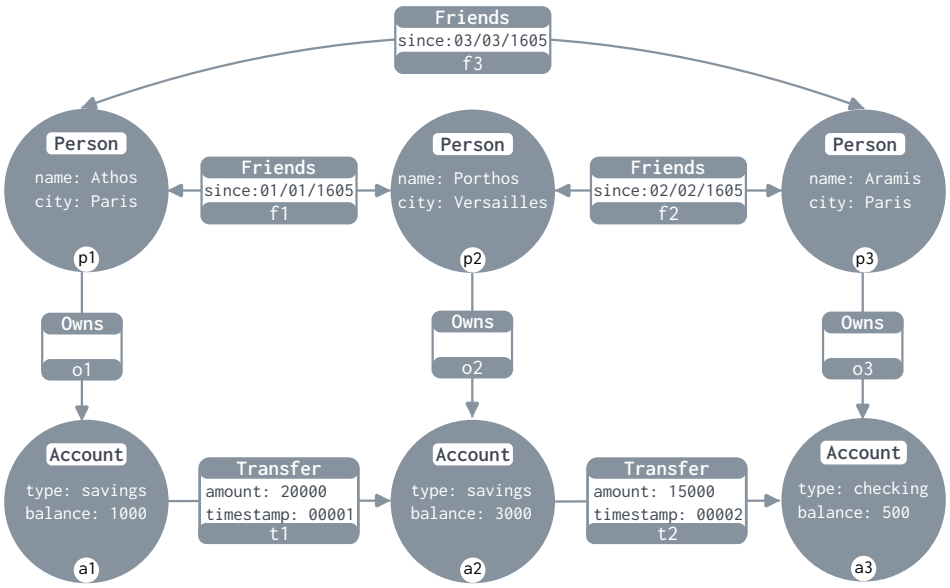


Figure 1: A labeled property graph

3 PATTERN MATCHING: TURNING PROPERTY GRAPHS INTO RELATIONS

We define property graphs and pattern matching, the key component of GQL and SQL/PGQ, that extracts relations from graphs.

3.1 Property Graphs

We use the standard definition (cf. [20]), and only consider directed edges. Assume pairwise disjoint countable sets $\mathcal{L}$ of labels, $\mathcal{K}$ of keys, Const of constants, N of node ids, and E of edge ids.

Definition 3.1 (Property Graph)**.** A property graph is a tuple $G = \langle N, E, \text{lab}, \text{src}, \text{tgt}, \text{prop}\rangle$ where

- $N \subset \mathsf{N}$ is a finite set of node ids used in G;
- $E \subset \mathsf{E}$ is a finite set of directed edge ids used in G;
- $\text{lab} : N \cup E \to 2^{\mathcal{L}}$ is a labeling function that associates with every node or edge id a (possibly empty) finite set of labels from $\mathcal{L}$;
- $\text{src}, \text{tgt} : E \to N$ define source and target of an edge;
- $\text{prop} : (N \cup E) \times \mathcal{K} \to \text{Const}$ is a partial function that associates a property value with a node/edge id and a key.

A *path* in G is an alternating sequence $u_0 e_1 u_1 e_2 \cdots e_n u_n$, for $n \geq 0$, of nodes and edges that starts and ends with a node and so that each edge e_i connects the nodes u_{i-1} and u_i for $i \leq n$. More precisely, for each $i \leq n$, either $\text{src}(e_i) = u_{i-1}$ and $\text{tgt}(e_i) = u_i$ (a *forward edge*), or $\text{src}(e_i) = u_i$ and $\text{tgt}(e_i) = u_{i-1}$ (a *backward edge*). Note that $n = 0$ is possible, in which case the path consists of a single node u_0. We shall explicitly spell out paths as $\text{path}(u_0, e_1, u_1, \cdots, e_n, u_n)$.

Two paths $p = \text{path}(u_0, e_0, \ldots, u_k)$ and $p' = \text{path}(u'_0, e'_0, \ldots, u'_j)$ *concatenate*, written as $p \odot p'$, if $u_k = u'_0$, in which case their *concatenation* $p \cdot p'$ is defined as $\text{path}(u_0, e_0, \ldots, u_k, e'_0, \ldots, u'_j)$. Note that a single-node path is a unit of concatenation: $p \cdot \text{path}(u)$ is defined iff $u = u_k$ and is equal to p.

3.2 Pattern Matching

Pattern matching is the key component of graph query languages. As already mentioned, an early abstraction of GQL and PGQ patterns was given in [20], but it retained too much of the baggage of the actual language (non-1NF outputs, nulls, a complex type system) for language analysis. Thus, here we refine the definitions from [20] to capture the core concepts of pattern matching, similarly to relational algebra for relational databases. To this end, we fix an infinite set Vars of variables and define Core GQL and Core PGQ pattern matching as follows:

$$\psi \;:=\; (x) \;\mid \overset{x}{\rightarrow} \mid \overset{x}{\leftarrow} \mid\; \psi_1\, \psi_2 \;\mid\; \psi^{n..m} \;\mid\; \psi\langle\theta\rangle \;\mid\; \psi_1 + \psi_2$$

where

- $x \in \text{Vars}$ and $0 \leq n \leq m \leq \infty$;
- variables x in node and edge patterns (x), $\overset{x}{\rightarrow}$, and $\overset{x}{\leftarrow}$ are optional,
- $\psi\langle\theta\rangle$ is a conditional pattern, and conditions are given by $\theta, \theta' := x.k = x'.k' \mid x.k < x'.k' \mid \ell(x) \mid \theta \vee \theta' \mid \theta \wedge \theta' \mid \neg\theta$ where $x, x' \in \text{Vars}$ and $k, k' \in \mathcal{K}$;
- $\psi_1 + \psi_2$ is only defined when their sets of free variables $\text{FV}(\psi_1)$ and $\text{FV}(\psi_2)$ are equal.

The sets of free variables are defined as follows:

- $\text{FV}\Big((x)\Big) = \text{FV}\left(\overset{x}{\rightarrow}\right) = \text{FV}\left(\overset{x}{\leftarrow}\right) := \{x\}$;
- $\text{FV}(\psi_1 + \psi_2) := \text{FV}(\psi_1)$
- $\text{FV}(\psi_1\, \psi_2) := \text{FV}(\psi_1) \cup \text{FV}(\psi_2)$
- $\text{FV}(\psi^{n..m}) := \emptyset$
- $\text{FV}(\psi\langle\theta\rangle) := \text{FV}(\psi)$

A pattern produces an output that consists of graph elements and their properties. Such an output Ω is a (possibly empty) tuple whose elements are either variables x or properties $x.k$. A *pattern with output*, or, pattern, for simplicity, is an expression ψ_Ω such that every variable present in Ω is in $\text{FV}(\psi)$.

Correspondence with Cypher and GQL. For the reader familiar with Cypher and/or GQL, we explain how our fomalization compares with these languages' patterns.

- (x) is a node pattern that binds the variable x to a node;
- $\xrightarrow{x}$ and $\xleftarrow{x}$ are forward edge and backward edge patterns, that also bind x to the matched edge;
- $\psi_1\,\psi_2$ is the concatenation of patterns,
- $\psi^{n..m}$ is the repetition of ψ between n and m times (with a possibility of $m = \infty$)
- $\psi\langle\theta\rangle$ corresponds to `WHERE` in patterns, conditions involve (in)equalities between property values, checking for labels, and their Boolean combinations;
- $\psi_1 + \psi_2$ is the union of patterns;
- ψ_Ω corresponds to the output forming clauses `RETURN` of Cypher and GQL and `COLUMNS` of SQL/PGQ, with Ω listing the attributes of returned relations.

Semantics. To define the semantics, we use set Values which is the union of Const $\cup$ N $\cup$ E. That is, its elements are node and edge ids, or values of properties, i.e., precisely the elements that can appear as outputs of patterns.

The *semantics of a path pattern* ψ, with respect to a graph G, is a set of pairs (p, μ) where p is a path and μ is a mapping FV $(\psi) \to$ Values. Recall that we write $\mu_\emptyset$ for the unique empty mapping with $\mathrm{dom}(\mu) = \emptyset$.

For the semantics of path patterns with output ψ_Ω we define $\mu_\Omega : \Omega \to$ Values as the projection of μ on Ω:

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

Full definitions are presented in Figure 2. For node and edge patterns with no variables, the mapping part of the semantics changes to $\mu_\emptyset$. The satisfaction of a condition θ by a mapping μ, written $\mu \models \theta$, is defined as follows: $\mu \models x.k = x'.k'$ if both $\mathrm{prop}(\mu(x), k)$ and $\mathrm{prop}(\mu(x'), k')$ are both defined and are equal (and likewise for $<$), and $\mu \models \ell(x)$ if $\ell \in \mathrm{lab}(\mu(x))$. It is then extended to Boolean connectives $\wedge, \vee, \neg$ in the standard way.

Pattern languages vs GQL and PGQ patterns. Compared to GQL and PGQ patterns as described in [15, 20, 21], we make some simplifications. First and foremost, they are to ensure that outputs of pattern matching are 1NF relations. Similarly to formal models of SQL by means of relational algebra and calculus over sets of tuples, we do not include bags, nulls, and relations whose entries are not atomic values. The differences manifest themselves in four ways.

First, we use set semantics rather than bag semantics. GQL and PGQ pattern can return tables with duplicate patterns; we follow their semantics up to multiplicity.

Second, in disjunctions $\psi_1 + \psi_2$ we require that free variables of ψ_1 and ψ_2 be the same. In GQL this is not the case, but then for a variable $x \in$ FV $(\psi_1) -$ FV (ψ_2), a match for ψ_2 would generate a *null* in the x attribute. Following the 1NF philosophy, we omit features that can generate nulls.

Third, repeated patterns $\psi^{n..m}$ have no free variables. In GQL and PGQ, free variables of ψ become *group variables* in $\psi^{n..m}$, leading to both a complex type system [20] and crucially non-flat outputs. Specifically such variables are evaluated to *lists*. For example, in the pattern $() \xrightarrow{x}{}^{0..\infty} ()$, the variable x would be mapped to the list of edge ids traversed by the path. These lists would be typically represented as values of an `ARRAY` type in implementations (cf. [39]) thus again violating 1NF.

Fourth, we do not impose any conditions on paths that can be matched. In GQL and PGQ they can be *simple* paths (no repeated nodes), or *trails* (no repeated edges), or *shortest* paths. In GQL, PGQ, and Cypher, paths themselves may be returned, and such restrictions therefore are necessary to ensure finiteness of output. Since we can only return graph nodes or edges, or their properties, we never have the problem of infinite outputs, and thus we chose not to overcomplicate the definition of core languages by deviating from flat tables as outputs.

In what *might* look like a simplification w.r.t. GQL and PGQ, we do not have explicit joins of patterns, i.e., ψ_1, ψ_2 with the semantics $[\![\psi_1, \psi_2]\!]_G = \{((p_1, p_2), \mu_1 \bowtie \mu_2) \mid (p_i, \mu_i) \in [\![\psi_i]\!]_G,\ i = 1, 2\}$. This is because such joins are definable with RA operations. This simplification is in the same spirit as not including the natural join in the definition of RA, since it can already be expressed with product, selection, and projection.

4 SQL/PGQ: THEORETICAL ABSTRACTIONS

Having defined patterns shared by SQL/PGQ and GQL, we can proceed to provide a theoretical abstraction of SQL/PGQ. Recall that in PGQ, the `MATCH` statement is embedded in `FROM`. In fact results of matches over a graph are simply treated as relations, or subqueries, over which the usual SQL can be asked. Taking again the view that our goal is to provide the core abstraction of the language, on top of which others can be built, we look at the essential core of relational languages, namely relational algebra. With this in mind, we define (for now informally):

Core PGQ = Relational Algebra over pattern matching outputs.

To make this definition formal, we define some very standard concepts. We assume an infinite countable set $\mathcal{S}$ of *relation symbols*, such that each $S \in \mathcal{S}$ is associated with a sequence $\mathrm{attr}(S) := A_1, \ldots, A_n$ of attributes for some $n > 0$; here attributes A_i come from a countably infinite set $\mathcal{A}$ of attributes. By $\mathrm{arity}(S)$ we mean n, and to be explicit about names of attributes write $S(A_1, \ldots, A_n)$.

Fix an infinite *domain* $\mathbf{U}$ of values. A *relation* over $S(A_1, \ldots, A_n)$ is a set of *tuples* $\mu : \{A_1, \ldots, A_n\} \to \mathbf{U}$. The domain of μ is denoted $\mathrm{dom}(\mu)$, and we often represent tuples as sets of pairs $(A, \mu(A))$ for $A \in \mathrm{dom}(\mu)$. A relational database $\mathcal{D}$ is a partial function that maps symbols $S \in \mathcal{S}$ to relations $\mathcal{D}(S)$ over S (if $\mathcal{D}$ is clear from the context we refer to $\mathcal{D}(S)$ simply as S, by a slight abuse of notation.) We say that $\mathcal{D}$ is over its domain $\mathrm{dom}(\mathcal{D})$.

Let μ be a tuple and $\boldsymbol{A} \subseteq \mathrm{dom}(\mu)$. We use $\mu \restriction \boldsymbol{A}$ to denote the restriction of μ to $\boldsymbol{A}$, that is, the mapping μ' with $\mathrm{dom}(\mu') = \boldsymbol{A}$ and $\mu'(A) := \mu(A)$ for every attribute $A \in \boldsymbol{A}$. Two tuples μ_1, μ_2 are *compatible*, denoted by $\mu_1 \sim \mu_2$, if $\mu_1(A) = \mu_2(A)$ for every $A \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$. For such compatible tuples define $\mu_1 \bowtie \mu_2$ as the mapping μ with $\mathrm{dom}(\mu) := \mathrm{dom}(\mu_1) \cup \mathrm{dom}(\mu_2)$, and $\mu(A) := \mu_1(A)$ if $A \in \mathrm{dom}(\mu_1)$ and $\mu(A) := \mu_2(A)$ otherwise.

If $A \in \mathrm{dom}(\mu)$ then the renaming $\rho_{A \to B}(\mu)$ of A to B is the mapping μ' with $\mathrm{dom}(\mu') = (\mathrm{dom}(\mu) \setminus \{A\}) \cup \{B\}$ where $\mu'(B) := \mu(A)$ and $\mu'(A') = \mu(A')$ for every other $A' \in \mathrm{dom}(\mu')$.

$$
\begin{aligned}
[\![(x)]\!]_G &:= \{(\mathsf{path}(n), \{x \mapsto n\}) \mid n \in \mathsf{N}\} \\
\left[\!\!\left[\overset{x}{\rightarrow}\right]\!\!\right]_G &:= \{(\mathsf{path}(n_1, e, n_2), \{x \mapsto e\}) \mid e \in \mathsf{E},\ \mathsf{src}(e) = n_1,\ \mathsf{tgt}(e) = n_2\} \\
\left[\!\!\left[\overset{x}{\leftarrow}\right]\!\!\right]_G &:= \{(\mathsf{path}(n_2, e, n_1), \{x \mapsto e\}) \mid e \in \mathsf{E},\ \mathsf{src}(e) = n_1,\ \mathsf{tgt}(e) = n_2\} \\
[\![\psi_1 + \psi_2]\!]_G &:= [\![\psi_1]\!]_G \cup [\![\psi_2]\!]_G \\
[\![\psi_1\, \psi_2]\!]_G &:= \left\{(p_1 \cdot p_2, \mu_1 \bowtie \mu_2) \;\middle|\; (p_1, \mu_1) \in [\![\psi_1]\!]_G,\ (p_2, \mu_2) \in [\![\psi_2]\!]_G,\ \mu_1 \sim \mu_2,\ p_1 \odot p_2\right\} \\
[\![\psi\langle\theta\rangle]\!]_G &:= \left\{(p, \mu) \in [\![\pi]\!]_G \mid \mu \models \theta\right\} \\
[\![\psi^{n..m}]\!]_G &:= \bigcup_{i=n}^{m} [\![\psi]\!]_G^i \text{ where} \\
&\qquad [\![\psi]\!]_G^0 := \left\{(\mathsf{path}(n), \mu_\emptyset) \;\middle|\; n \in \mathsf{N}\right\} \\
&\qquad [\![\psi]\!]_G^n := \left\{(p_1 \cdots p_n, \mu_\emptyset) \mid \exists \mu_1, \ldots, \mu_n : (p_i, \mu_i) \in [\![\psi]\!]_G \text{ and } p_i \odot p_{i+1} \text{ for all } i < n\right\},\ n > 0 \\
[\![\psi_\Omega]\!]_G &:= \left\{\mu_\Omega \mid \exists p : (p, \mu) \in [\![\psi]\!]_G\right\}
\end{aligned}
$$

Figure 2: Semantics of patterns and patterns with output

Relational Algebra (RA). We use a standard presentation of RA. Given a schema S which is a finite subset of $\mathcal{S}$, the expressions Q of RA(S) and selection conditions θ are defined as

$$
\begin{aligned}
Q, Q' &:= R \mid \pi_{\boldsymbol{A}}(Q) \mid \sigma_\theta(Q) \mid Q \bowtie Q' \mid Q \cup Q' \mid Q - Q' \\
\theta &:= A = A' \mid \neg\theta \mid \theta \vee \theta \mid \theta \wedge \theta
\end{aligned}
$$

where R ranges over relations in S. The sets of attributes of expressions $\mathsf{attr}(Q)$ are defined by extending $\mathsf{attr}(R)$, namely $\mathsf{attr}(\pi_{\boldsymbol{A}}(Q))$ is $\boldsymbol{A}$, while both of $\mathsf{attr}(\sigma_\theta(Q))$ and $\mathsf{attr}(Q \circ Q')$ are $\mathsf{attr}(Q)$, for $\circ$ being union and difference; $\mathsf{attr}(Q \bowtie Q') = \mathsf{attr}(Q) \cup \mathsf{attr}(Q')$ and $\mathsf{attr}(\rho_{A\to A'}(Q)) = (\mathsf{attr}(Q) \setminus \{A\}) \cup \{A'\}$.

The expressions of RA must satisfy the usual well-definedness rules: $\pi_{\boldsymbol{A}}(Q)$ is well-defined if $\boldsymbol{A} \subseteq \mathsf{attr}(Q)$; set operations are defined if $\mathsf{attr}(Q) = \mathsf{attr}(Q')$, and for renaming from A to A' we must have $A \in \mathsf{attr}(Q)$ and $A' \notin \mathsf{attr}(Q)$.

The result of evaluation of a query Q on a database $\mathcal{D}$ is a relation $[\![Q]\!]_{\mathcal{D}}$ over $\mathsf{attr}(Q)$ defined as:

$$
\begin{aligned}
[\![R]\!]_{\mathcal{D}} &:= \mathcal{D}(R) \\
[\![\pi_{\boldsymbol{A}}(Q)]\!]_{\mathcal{D}} &:= \{\mu \restriction_{\boldsymbol{A}} \mid \mu \in Q\} \\
[\![\sigma_\theta(Q)]\!]_{\mathcal{D}} &:= \{\mu \mid \mu \in Q \text{ and } \mu \models \theta\} \\
[\![\rho_{A\to A'}(Q)]\!]_{\mathcal{D}} &:= \{\rho_{A\to A'}(\mu) \mid \mu \in Q\} \\
[\![Q \bowtie Q']\!]_{\mathcal{D}} &:= \{\mu \bowtie \mu' \mid \mu \in Q,\ \mu' \in Q'\} \\
[\![Q \circ Q']\!]_{\mathcal{D}} &:= [\![Q]\!]_{\mathcal{D}} \circ [\![Q']\!]_{\mathcal{D}} \qquad \text{for } \circ \in \{\cup, \setminus\}
\end{aligned}
$$

with $\mu \models \theta$ having the standard semantics $\mu \models A = A'$ iff $A, A' \in \mathsf{dom}(\mu)$ and $\mu(A) = \mu(A')$, extended to Boolean connectives $\wedge, \vee, \neg$.

Core PGQ. Assume that for each variable $x \in \mathsf{Vars}$ and each key $k \in \mathcal{K}$, both x and $x.k$ belong to the set of attributes $\mathcal{A}$. For each pattern ψ and each output specification Ω, we have a relation symbol $R_{\psi,\Omega}$ whose set of attributes are the elements of Ω. Let Pat contain all such relation symbols.

Definition 4.1 (Core PGQ)**.** *Core PGQ* is defined as RA(Pat), i.e., the set of relational algebra expressions over the schema Pat.

To define the semantics of Core PGQ queries, assume without loss of generality that $\mathsf{Values} \subseteq \mathbf{U}$. This ensures that results of pattern matching are relations of the schema Pat, because for every path pattern with output ψ_Ω and a property graph G, the table $[\![\psi_\Omega]\!]_G$ is an instance of relation $R_{\psi,\Omega}$ from Pat. Then the semantics of Core PGQ is simply the extension of the semantics of RA defined above where for base relations we have $\left[\!\left[R_{\psi,\Omega}\right]\!\right]_G := [\![\psi_\Omega]\!]_G$.

5 GQL: THEORETICAL ABSTRACTIONS

We next provide a formal model of GQL. Recall that it shares patterns with PGQ. What is different is the way GQL processes results of pattern matching: not in a bottom-up way with RA operators like PGQ, but rather in sequential, or pipelined way where the output of each operation in a sequence serves as the input for the next operation. Using terminology adopted by Cypher [22], GQL calls this *linear composition*. Unlike RA, it lacks proper formalization, and thus next we provide a formal description of a different flavor of RA, obtained by linear composition.

5.1 LCRA: Linear Composition RA

This language captures the sequential (linear) application of relational operators as seen in Cypher, GQL, and also PRQL. Its expressions over a schema S, denoted by LCRA(S), are defined as:

$$
\begin{aligned}
\textit{Linear Clause:}\quad & \mathbf{L}, \mathbf{L}' := S \mid \pi_{\boldsymbol{A}} \mid \sigma_\theta \mid \rho_{A\to A'} \mid \mathbf{L}\,\mathbf{L}' \mid \{Q\} \\
\textit{Query:}\quad & Q, Q' := \mathbf{L} \mid Q \cap Q' \mid Q \cup Q' \mid Q \setminus Q'
\end{aligned}
$$

where S ranges over S, while $\boldsymbol{A} \subseteq \mathcal{A}$, and $A, A' \in \mathcal{A}$, and θ is defined as for RA. Unlike for RA, the output schema of LCRA clauses and queries can be determined only dynamically.

The semantics $[\![\,]\!]_{\mathcal{D}}$ of LCRA clauses $\mathbf{L}$ and queries Q is a *mapping* from relations into relations (known as *driving tables* for Cypher and GQL). It is defined as follows:

$$
\begin{aligned}
[\![S]\!]_{\mathcal{D}}(\mathbf{R}) &:= \mathbf{R} \bowtie \mathcal{D}(S) \\
[\![\pi_{\boldsymbol{A}}]\!]_{\mathcal{D}}(\mathbf{R}) &:= \{\mu \restriction_{\boldsymbol{A} \cap \mathsf{attr}(\mathbf{R})} \mid \mu \in \mathbf{R}\} \\
[\![\sigma_\theta]\!]_{\mathcal{D}}(\mathbf{R}) &:= \{\mu \mid \mu \in \mathbf{R},\ \mu \models \theta\} \\
[\![\rho_{A\to A'}]\!]_{\mathcal{D}}(\mathbf{R}) &:= \{\rho_{A\to A'}(\mu) \mid \mu \in \mathbf{R},\ A' \notin \mathsf{dom}(\mu)\} \\
[\![\mathbf{L}\mathbf{L}']\!]_{\mathcal{D}}(\mathbf{R}) &:= [\![\mathbf{L}']\!]_{\mathcal{D}}([\![\mathbf{L}]\!]_{\mathcal{D}}(\mathbf{R})) \\
[\![\{Q\}]\!]_{\mathcal{D}}(\mathbf{R}) &:= \mathbf{R} \bowtie [\![Q]\!]_{\mathcal{D}}(\mathbf{R}) \\
[\![Q \circ Q']\!]_{\mathcal{D}}(\mathbf{R}) &:= [\![Q]\!]_{\mathcal{D}}(\mathbf{R}) \circ [\![Q']\!]_{\mathcal{D}}(\mathbf{R}), \qquad \text{for } \circ \in \{\cup, \cap, -\}
\end{aligned}
$$

A clause or a query of LCRA always looks at a unique input relation **R**. If a clause is a name of a relation S or an entire query $\{Q\}$, then it is joined with **R**. Projection, selection, and renaming clauses behave in the usual way, and apply to the input relation **R**. The meaning of set operations – union, intersection, difference – is also standard, and two clauses $\mathbf{LL}'$ are simply executed in sequence, with the result of **L** on **R** becoming the input to $\mathbf{L}'$.

This semantics determines (dynamically) the set of attributes of outputs of clauses and queries; for completeness we present it here:

$$\begin{aligned}
\mathrm{attr}(\llbracket S \rrbracket_{\mathcal{D}}(\mathbf{R})) &:= \mathrm{attr}(S) \cup \mathrm{attr}(R) \\
\mathrm{attr}(\llbracket \pi_{\mathbf{A}} \rrbracket_{\mathcal{D}}(\mathbf{R})) &:= \mathbf{A} \cap \mathrm{attr}(\mathbf{R}) \\
\mathrm{attr}(\llbracket \sigma_{\theta} \rrbracket_{\mathcal{D}}(\mathbf{R})) &:= \mathrm{attr}(\mathbf{R}) \\
\mathrm{attr}(\llbracket \rho_{A\to A'} \rrbracket_{\mathcal{D}}(\mathbf{R})) &:= \mathrm{attr}(\mathbf{R}) \setminus \{A'\} \cup \{A\} \\
\mathrm{attr}(\llbracket \{Q\} \rrbracket_{\mathcal{D}}(\mathbf{R})) &:= \mathrm{attr}(\llbracket Q \rrbracket_{\mathcal{D}}(\mathbf{R})) \cup \mathrm{attr}(\mathbf{R}) \\
\mathrm{attr}(\llbracket Q \circ Q' \rrbracket_{\mathcal{D}}(\mathbf{R})) &:= \mathrm{attr}(\llbracket Q \rrbracket_{\mathcal{D}}(\mathbf{R}))
\end{aligned}$$

For producing the output of a query Q on a database $\mathcal{D}$ we need a starting value of the driving table **R**, and this is taken to be $I_\emptyset$, the relation containing only one empty tuple $\mu_\emptyset$ where $\mathrm{dom}(\mu_\emptyset) := \emptyset$, cf. [21, 22]. Then query output on a database is defined as $\llbracket Q \rrbracket_{\mathcal{D}}(I_\emptyset)$.

5.2 Core GQL

Just as we defined Core PGQ as RA over output of patterns, we now define Core GQL as LCRA over the same.

Definition 5.1 (Core GQL)**.** The language *Core GLQ* is defined as the set of linear composition relational algebra expressions over the schema Pat, i.e., LCRA(Pat).

As with Core PGQ, we define the semantics of Core GQL by $\llbracket R_{\psi,\Omega} \rrbracket_G := \llbracket \psi_\Omega \rrbracket_G$ and then use the semantic of LCRA above.

We now present an example to explain how GQL's linear composition works. We use a simplified query based on the money laundering query from the introduction. It looks for someone who has two friends in a city different from theirs, and outputs the person's name and account (`Porthos` and `a2` in our example):

```
MATCH (x)-[:Friends]->(y)-[:Friends]->(z),
(y)-[:Owns]->(acc_y)
FILTER (y.city) <> (x.city)
AND (x.city=z.city)
RETURN y.name AS name, acc_y AS account
```

The equivalent Core GQL formula is

$$\rho_{acc\_y\to\text{account}}\ \rho_{y.\text{name}\to\text{name}}\ \pi_{y.\text{name},acc\_y}\ \sigma_{y.\text{city}\neq x.\text{city}\wedge x.\text{city}=z.\text{city}}\ R_{\psi_1,\Omega_1}\ R_{\psi_2,\Omega_2}$$

where

$$\begin{aligned}
\psi_1 &:= \big((x) \xrightarrow{e_1} (y)\ (y) \xrightarrow{e_2} (z)\big)\langle \mathrm{Friends}(e_1) \wedge \mathrm{Friends}(e_2)\rangle \\
\psi_2 &:= \big((y) \xrightarrow{e_3} (\mathrm{acc_}y)\big)\langle \mathrm{Owns}(e_3)\rangle \\
\Omega_1 &:= (x,\ y,\ z,\ x.\mathrm{city},\ y.\mathrm{city},\ z.\mathrm{city}) \\
\Omega_2 &:= (y,\ \mathrm{acc_}y)\,.
\end{aligned}$$

5.3 Equivalence of Core PGQ and Core GQL

We say that two queries Q_1, Q_2 (possibly from different languages) are *equivalent* if $\llbracket Q_1 \rrbracket_{\mathcal{D}} = \llbracket Q_2 \rrbracket_{\mathcal{D}}$ for every database $\mathcal{D}$. A query language L_1 is *subsumed by* L_2 if for each query Q_1 in L_1 there is an equivalent query Q_2 in L_2. If there is also a query $Q_2 \in L_2$ for which there is no equivalent query $Q_1 \in L_1$ then L_1 is said to be *strictly less expressive than* L_2. Finally L_1 and L_2 are *equivalent* if L_1 is subsumed by L_2 and L_2 is subsumed by L_1.

Theorem 5.1. *Languages* RA(S) *and* LCRA(S) *are equivalent, for every schema* S

Thus, LCRA proposed as the relational processing engine of graph languages like Cypher and GQL is the good old RA in a slight disguise. As an immediate consequence of Theorem 5.1 we have:

Corollary 5.2. *The languages Core PGQ and Core GQL have the same expressive power.*

Notice that the definition of linear clauses and queries of LCRA are mutually recursive as we can feed any query Q back into clauses via $\{Q\}$ (this corresponds to the `CALL` feature of Cypher and GQL). If this option is removed, and linear clauses are not dependent on queries, we get a simplified language sLCRA (simple LCRA). Specifically, in this language linear clauses are given by the grammar $\mathbf{L} := S \mid \pi_A \mid \sigma_\theta \mid \rho_{A\to A'} \mid \mathbf{LL}$. To see why the $\{Q\}$ clause was necessary in the definition of LCRA, we show

Proposition 5.3. sLCRA *is strictly less expressive than* LCRA.

5.4 The origins of linear composition

Linear composition features prominently in graph languages (Cypher, GQL) and also some relational languages (PRQL); at the same time it had not been formalized nor studied as its non-linear relational algebra analog (until now). Thus we use this short section to briefly explain the origins of linear composition. Linear composition was introduced in the design of Cypher [22] as a way to bypass the lack of a compositional language for graphs. Specifically, pattern matching transforms graphs into relational tables, and other Cypher operations modify these tables. If we have two such read-only queries $G \to \mathbf{T}_1$ and $G \to \mathbf{T}_2$ from graphs to relational tables, it is not clear how to compose them. To achieve composition, Cypher read-only queries are of the form $Q : G \times \mathbf{T} \to \mathbf{T}'$ (and its read/write queries are of the form $Q : G \times \mathbf{T} \to G' \times \mathbf{T}'$). In other words, they turn a graph *and a table* into a table. Thus, the composition of two queries $Q_1, Q_2 : G \times \mathbf{T} \to \mathbf{T}'$ is their *linear composition* $Q_1\ Q_2$ which on a graph G and table $\mathbf{T}$ returns $Q_2(G, Q_1(G, \mathbf{T}))$.

Independently, the same approach was adopted by a relational language PRQL [33], where P stands for "pipelined" but the design philosophy is identical. For example one could write

```
FROM R FILTER A=1 JOIN:INNER S FILTER B=2 SELECT C, D
```

with each clause applied to the output of the previous clause. The above query is the same as relational algebra query

$$\pi_{C,D}\Big(\sigma_{B=2}\big(\sigma_{A=1}(R) \bowtie S\big)\Big)$$

Though PRQL design is relations-to-relations, the motivation for pipelined or linear composition comes from creating a database analog of `dplyr` [41], a data manipulation library in R, that can be translated to SQL. While `dplyr`'s operations are very much relational in spirit, it is integrated into a procedural language, and

hence the imperative style of programming was inherited by PRQL and also adopted by the piped syntax of SQL [37].

To recap, we defined simple theoretical abstractions *Core PGQ* of SQL/PGQ and *Core GQL* of GQL, that share the same pattern matching language turning graphs into relations; then on top of pattern matching outputs we use RA for PGQ and LCRA for GQL. It should be kept in mind that these abstractions capture the essense of SQL/PGQ and GQL in the same way as RA and first-order logic capture the essence of SQL: they define a theoretical core that is amenable to a formal study, but real languages, be it 500 pages of the GQL standard or over 4000 pages of the SQL standard, have many more features.

6 LIMITATIONS OF PATTERNS: A THEORETICAL INVESTIGATION

A common pattern of query language development is this: first, a small core language is designed; its focus is on declarativeness and optimizability. As a consequence the expressiveness of such a language is limited. These limitations are typically observed in practice by programmers' inability to write certain queries and often later proven formally, for a theoretical language that defines the key features of the practical one. Using these inputs as motivations, extra features are added to the real-life language if practical applications warrant this and user demands.

A classical example of this development cycle is SQL and recursion. In this case, we had a "canonical" query whose inexpressibility it was important to show in order to justify additions to the language. The query was transitive closure of relation (or, equivalently, testing for graph connectivity [4, 18, 23]). Thus, before embarking on the study of the expressiveness of GQL and SQL/PGQ, we ask for an analog of such "canonical" queries for them that users want to express but appear to be unable to.

Fortunately, we have such queries, thanks to the discussions already happening in the ISO committee that maintains both SQL and GQL standards [31, 43]. In fact much of recent effort tries to repair what appears to be a hole in the expressiveness of the language. Specifically, it seems to be impossible to express queries that impose conditions on how values of edges properties change along the paths, even if the same conditions can be expressed for properties of nodes. To give a simple example of such a condition, consider the following:

- we *can* check, by a very simple pattern, if there is a path of transfers between two accounts where balance in intermediary accounts (values held in *nodes*) increases, but
- it appears that we *cannot* check if there is a path of transfers between two accounts where timestamp (values held in *edges*) increases.

This perceived deficiency has already led to early proposals to significantly enhance pattern matching capabilities of the language [31, 43], in a way whose complexity implications appear significant but are not fully understood.

Before such a dramatic expansion of the language gets a stamp of approval, it would be nice to know whether it is actually needed. The goal of this section is to provide both theoretical and experimental evidence that we do need extra language features to express queries such as *"increasing values in edges"*.

6.1 What *can* be expressed

A pattern ψ can be viewed as a query $\big((x_s)\ \psi\ (x_t)\big)_{x_s,x_t}$ that returns endpoints (source and target) of the paths matched by ψ as values of attributes x_s and x_t. Earlier queries are defined formally as:

- $Q^{\mathsf{N}}_{\uparrow}$ returns endpoints of a path along which the value of property k of *nodes* increases. It returns pairs (u_0, u_n) of nodes such that there is path $\mathrm{path}(u_0, e_1, u_1, \ldots, e_n, u_n)$ so that $u_0.k < u_1.k < \cdots < u_n.k$.
- $Q^{\mathsf{E}}_{\uparrow}$ returns endpoints of a path along which the value of property k of *edges* increases. It returns pairs (u_0, u_n) of nodes such that there is path $\mathrm{path}(u_0, e_1, u_1, \ldots, e_n, u_n)$ so that $e_1.k < e_2.k < \cdots < e_n.k$.

It is a simple observation that $Q^{\mathsf{N}}_{\uparrow}$ is expressible by the Core PGQ and GQL pattern with output

$$(x_s)\ \Big(\ \big((x) \to (y)\langle x.k < y.k\rangle\big)^{0..\infty}\ \Big)\ (x_t).$$

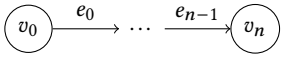
In fact this can be generalized to *order motifs* which are defined as strings over the alphabet $\{\uparrow, \downarrow\}$. Given such a string w, the query Q^{N}_{w} matches paths which can be decomposed according to w so that in each segment corresponding to $\uparrow$ values of property k increase and in each segment corresponding to $\downarrow$ these values decrease. For example, $Q^{\mathsf{N}}_{\uparrow\downarrow\uparrow}$ matches arbitrary length paths where values of property k of nodes first increases, then decreases, and then increases again. With the same approach as the above query, we can see that Q^{N}_{w} is expressible for every order motif w.

What happens when we move from nodes to edges? The same approach will not work for $Q^{\mathsf{E}}_{\uparrow}$: if we (erroneously) try to express it by

$$\big(\ ()\xrightarrow{x}()\xrightarrow{y}()\langle x.k < y.k\rangle\ \big)^{0..\infty}$$

it fails: on the input $()\xrightarrow{3}()\xrightarrow{4}()\xrightarrow{1}()\xrightarrow{2}()$ (where the numbers on the edges are values of property k) it returns the start and the end node of the path, even though values in edges do not increase. This is due to the semantics of path concatenation: two paths concatenate if the last node of the first path equals the first node of the second, and therefore conditions on edges in two concatenated or repeated paths are completely "local" to those paths.

In some cases, where graphs are of special shape, we can obtain the desired patterns by resorting to tricks that a normal query language user would be rather unlikely to find. Specifically, consider property graphs whose underlying graph structures are just paths. That is, we look at *annotated paths* P_n, which are of the form



where $v_0, \ldots, v_n$ are distinct nodes, $e_0, \ldots, e_{n-1}$ are distinct edges (for $n > 0$), and each edge e_i has the property $e_i.k$ defined. Then define the pattern $\psi^{\mathsf{E}}_{<}$ as

$$\begin{aligned} &(x_s)\ \Big(\big((u)\xrightarrow{x}(z)\xrightarrow{y}(v)\xleftarrow{w}(z)\big)\langle x.k < y.k\rangle\Big)^{1..\infty} \to (x_t)\\ +\quad &(x_s)\to(x_t)\end{aligned}$$

A pattern with output ψ_Ω expresses a query Q if $[\![\psi_\Omega]\!]_G = Q(G)$ for every graph G.

Proposition 6.1. *The pattern* $\big(\psi^{\mathsf{E}}_{<}\big)_{(x_s,x_t)}$ *expresses* $Q^{\mathsf{E}}_{\uparrow}$ *on annotated paths.*

In the first disjunct, the forward edge $\xrightarrow{y}$ serves as a look-ahead that enables checking whether the condition holds, and the backward edge $\xleftarrow{w}$ enables us to continue constructing the path in the next iteration from the correct position (z). The fact that the query operates on P_n ensures that the last edge to x_t does not violate the condition of the query as it was already traversed in the iterated subpattern. The second disjunct takes care of paths of length 1.

6.2 What *cannot* be expressed with patterns

To achieve expressibility of $Q^{\mathsf{E}}_{\uparrow}$ in Proposition 6.1 we made two strong assumptions: not only is the input of a very special shape (a directed path with forward edges), but also the pattern uses backwards edges; the latter would be natural for an *oriented* path (in which edges can go in either direction [26]), but looks rather unnatural in this setting. Thus, we ask ourselves whether $Q^{\mathsf{E}}_{\uparrow}$ can be expressed in a *natural* way. In fact we can pose a more general question about queries Q^{E}_{w} for arbitrary order motifs; in Section 6.1 we saw that on nodes, all order motifs can be expressed.

To formalize what we mean by "natural", define *one-way path patterns* by a restriction of the grammar:

$$\psi \;:=\; (x) \;\mid \xrightarrow{x} \mid\; \psi_1 + \psi_2 \;\mid\; \psi^{n..m} \;\mid\; \psi_{\langle\theta\rangle} \;\mid\; \psi_1\,\psi_2$$

where we require that $\mathsf{FV}(\psi_1) \cap \mathsf{FV}(\psi_2) = \emptyset$ in $\psi_1\,\psi_2$ (and variables x, as before, are optional). The omission of backward edges $\xleftarrow{x}$ in one-way patterns is quite intuitive. The restriction on variable sharing in concatenated patterns is because backward edges can be simulated by simply repeating variables, as done above in $\psi^{\mathsf{E}}_{<}$.

If there is a natural way, easily found by programmers, of writing the $Q^{\mathsf{E}}_{\uparrow}$ query in PGQ and GQL, one would expect it to be done without backward edges. Yet, this is not the case. In fact, no order motif on edges can be captured by GQL and PGQ patterns.

Theorem 6.2. *There is no order motif w such that Q^{E}_{w} is expressible by a one-way path pattern query.*

The inexpressibility of $Q^{\mathsf{E}}_{\uparrow}$ is then an immediate corollary for $w = \uparrow$. This result is a direct consequence of a general pumping argument applied to annotated paths accepted by one-way patterns.

Theorem 6.3. *For every one-way path pattern ψ, if for every $n \in \mathbb{N}$ there exists an annotated path p of length n accepted by ψ then there exists $n_0 \in \mathbb{N}$ such that for every annotated path p accepted by ψ with $|p| > n_0$, the path p can be decomposed as $p = p_1p_2p_3$, $|p_2| > 1$ and for every $n \in \mathbb{N}$, the annotated path $p_1p_2^{n+1}p_3$ is also accepted by ψ.*

Note that in the newly constructed path with repetitions, we assign new ids to the different occurrences of the elements in p_2, while keeping the data (annotations) unchanged. The idea behind the proof is as follows: Since, by definition, there are infinitely many annotated paths that conform to ψ, it must exhibit unbounded repetition. However, because the semantics disregard variables that occur within unbounded repetitions, the transfer of information between iterations is limited. This restriction allows us to repeat parts of the annotated path while preserving the same semantics and, consequently, still conforming to ψ.

We can derive a further corollary of this theorem that reinforces the intuition that GQL and PGQ patterns are incapable of capturing data values along a path "as a whole."

A canonical example of such a condition is checking whether all values of a specific property of nodes or edges along a path are distinct. Formally, we define:

- $Q^{\mathsf{N}}_{\neq}$ returns pairs (u_0, u_n) of nodes such that there is path $\mathsf{path}(u_0, e_1, u_1, \ldots, e_n, u_n)$ so that $u_i.k \neq u_j.k$ for all $0 \leq i < j \leq n$.

and $Q^{\mathsf{E}}_{\neq}$ is defined likewise but for edges in place of nodes. One-way path patterns do not have enough power to express such queries.

Corollary 6.4. *No one-way path pattern expresses $Q^{\mathsf{N}}_{\neq}$ nor $Q^{\mathsf{E}}_{\neq}$.*

6.3 What *cannot* be expressed in full GQL

Having examined the expressiveness of patterns, we now look at the entire query languages Core GQL and Core PGQ and compare them with recursive SQL and linear Datalog. Recursive SQL is a good comparison target as the most natural relational language into which graph queries involving pattern with arbitrary length paths are translated [38, 42]. The theoretical basis for recursive SQL common table expressions is *linear Datalog*, i.e., the fragment of Datalog in which definitions can refer to recursively defined predicates at most once. This is precisely the restriction of recursive SQL: a recursively defined table can appear at most once in `FROM`.

Of course recursive SQL can be enormously powerful: as already mentioned, combining recursion and arithmetic/aggregates one can simulate Turing machines [1]. Thus, to make the comparison fair, we look at *positive recursive SQL*: this is the fragment of recursive SQL where subqueries can only define equi-joins. In other words, they only use conjunctions of equalities in `WHERE`. Such a language just adds recursion on top of unions of conjunctive queries, and ensures termination and tractable data complexity [6]. We show that even these simple fragments of SQL and Datalog can express queries that Core GQL and PGQ cannot define.

Theorem 6.5. *There are queries that are expressible in positive recursive SQL, and in linear Datalog, and yet are not expressible in Core GQL nor Core PGQ.*

At the end of the section, we explain why this is quite surprising in view of what we know about Core GQL and PGQ: complexity-theoretic considerations strongly suggest these should define *all* queries from linear Datalog, and yet this is not the case due to subtle deficiencies in the language design, which we outline in Section 8.

To talk about expressing property graph queries in relational languages, we must represent graphs as relations. There are many possibilities, and it does not matter (for showing Theorem 6.5) which one we choose, as these different representations are inter-definable by means of unions of select-project-join queries. Since graphs in the separating example have labels and do not have property values, we use an encoding consisting of unary relations N_ℓ storing ℓ-labeled nodes, and binary relations E_ℓ storing pairs of nodes (n_1, n_2) with an ℓ-labeled edge between them.

To sketch the idea of the separating query, we define *dataless paths* as graphs G_n, $n > 0$, with nodes $v_0, \ldots, v_n$, edges (v_0, v_1), $(v_1, v_2), \ldots, (v_{n-1}, v_n)$, where v_0 has label *min_elt* and v_n has label *max_elt*. The separating query asks: is n a power of 2?

The inexpressibility proof of this is based on showing that GQL queries can only define Presburger properties of lengths of dataless paths, by translating Core GQL queries on such paths into formulae

of Presburger Arithmetic (i.e., the first-order theory of $\langle \mathbb{N}, +, < \rangle$). Therefore the only definable properties of lengths are semilinear sets [24]. Semi-linear subsets of $\mathbb{N}$ are known to be ultimately periodic: that is, for such a set $S \subseteq \mathbb{N}$, there exists a threshold t and a period p such that $n \in S$ if and only if $n + p \in S$, as long as $n > t$. Clearly the set $\{2^k \mid k \in \mathbb{N}\}$ is not such.

The query can be expressed in positive recursive SQL:

```
WITH RECURSIVE ADD(A, B, C) AS
  ( (SELECT P.A, MIN_ELT.A AS B, P.A AS C
     FROM P, MIN_ELT)
   UNION
    (SELECT MIN_ELT.A, P.A AS B, P.A AS C
     FROM P, MIN_ELT)
   UNION
    (SELECT ADD.A, P1.B, P2.B AS C
     FROM ADD, P P1, P P2
     WHERE ADD.B=P1.A AND ADD.C=P2.A) ),
  POW2(A, B) AS
  ( (SELECT P2.A, P2.B
     FROM MIN_ELT M, P P1, P P2
     WHERE M.A=P1.A AND P1.B=P2.A)
  UNION
    (SELECT P.B AS A, ADD.C AS B
     FROM POW2, ADD, P
     WHERE POW2.A=P.A AND ADD.A=POW2.B
           AND ADD.B=POW2.B) )
(SELECT 'YES' FROM POW2, MAX_ELT
 WHERE POW2.B=MAX_ELT.A)
```

The path is given by a binary relation P containing pairs (v_i, v_{i+1}) while minimal/maximal elements v_0 and v_n are given by unary relations MIN_ELT and MAX_ELT. The first common table expression defines a ternary relation ADD with tuples (v_i, v_j, v_k) such that $i+j = k$. If v_i is in MIN_ELT, then (v_i, v_j, v_j) and (v_j, v_i, v_j) are in ADD for all j (the basis of recursion), and if (v_i, v_j, v_k) is in ADD then so is (v_{i+1}, v_j, v_{k+1}) (the recursive step). After that POW2 builds a relation with tuples (v_i, v_j) for $j = 2^i$. Indeed, if (v_i, v_j) is in POW2, then so is (v_{i+1}, v_k) for $k = 2 \cdot j$, which is tested by $(v_j, v_j, v_k) \in$ ADD. The length of the path is a power of 2 if the second projection of POW2 contains MAX_ELT.

Note that ADD is defined by a linear Datalog program and POW2 is defined by a linear Datalog program that uses ADD as EDB. Hence, the entire query is defined by a piece-wise linear program (where already defined predicates can be used as if they were EDBs). It is known that such programs can be expressed in linear Datalog [3].

Complexity-theoretic considerations. We now explain why the inexpressibility result is quite unexpected. Note that all graph pattern languages can express the reachability query. It is complete for the complexity class NLogSpace via first-order reductions [28]: an extension of first-order logic with the reachability predicate capturtes precisely all NLogSpace queries. Since GQL and PGQ can emulate relational algebra (and thus first-order logic) over pattern matching results, it appears that they should be able to express all NLogSpace queries. In fact graph query languages expressing all NLogSpace queries have been known for a long time, starting with GraphLog [13], which introduced the ubiquitous notion of CRPQs.

However, Theorem 6.5 not only refutes the complexity-based intuition, but in fact the separating query has an even lower DLogSpace complexity. Indeed, one can traverse the dataless path graph while maintaining the counter that needs a logarithmic number of bits. This limitation highlights a deficiency of the language design: despite having access to reachability and full power of first-order logic on top of it, Core GQL and PGQ fall short of a declarative language that is first-order logic with the reachability predicate. The reason for this deficiency, that should ideally be addressed in future revisions of the standards, is discussed below in Section 8.

7 LIMITATIONS OF PATTERNS: AN EXPERIMENTAL INVESTIGATION

We have shown that many queries tracing changes in property values of edges cannot be expressed in Core GQL and PGQ, among them the simple query $Q^{\mathsf{E}}_{\uparrow}$ that generated significant interest in the GQL standardization committee of ISO [31, 43]. Of course real-life languages have more expressiveness than their theoretical counterparts, and thus real-life GQL, SQL/PGQ, and also Cypher can express this query. However, they do so in a rather convoluted way. We now show experimentally that this way of expressing simple graph queries has no realistic chance to work, as it generates enormous (exponential size) intermediate results, and the query would not terminate even on *tiny graphs*.

The idea of expressing $Q^{\mathsf{E}}_{\uparrow}$ is that its *complement* is easily definable in Core GQL. In GQL, PGQ, and Cypher, paths can be named and output. This is an advanced feature that we omitted in our core language: since paths are represented as lists, it results in non-flat outputs. The complement of $Q^{\mathsf{E}}_{\uparrow}$ is expressed by the pattern $\psi^{\neg}$:

```
(v1)->*(-[x]-> -[y]-> WHERE x.k>=y.k)->*(v2)
```

testing for a pair of consecutive edges that break the increasing motif; it can also be expressed by CoreGQL as

$$\psi^{\neg} := (v_1) \rightarrow^{0..\infty} \big(() \xrightarrow{x} () \xrightarrow{y} () \big) \langle x.k \geq y.k \rangle \rightarrow^{0..\infty} (v_2)\,.$$

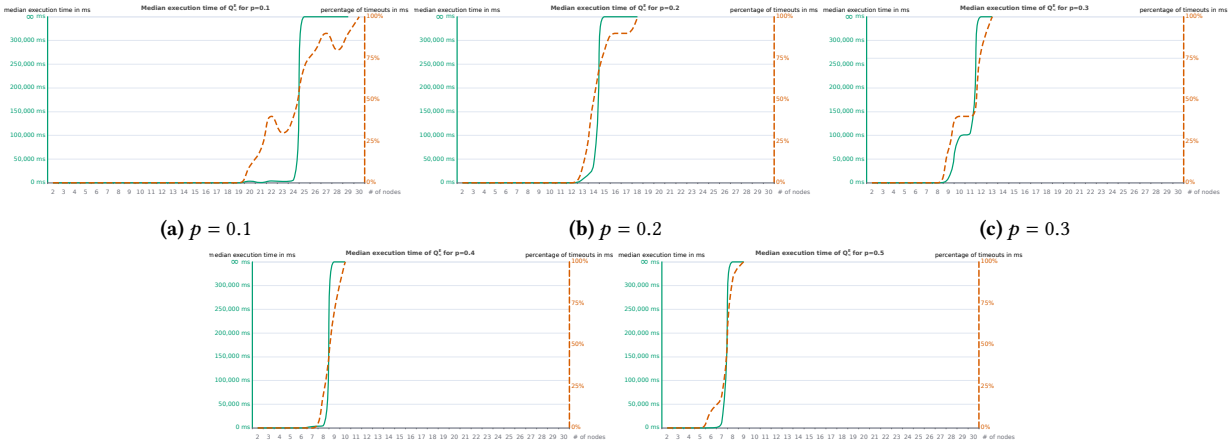
Thus, the query below

```
MATCH p = (v1) ->* (v2) RETURN v1, v2, p
EXCEPT
MATCH p=φ¬  RETURN v1, v2, p
```

finds all nodes v_1, v_2 and path p between them satisfying $Q^{\mathsf{E}}_{\uparrow}$. The mere fact of expressing something does not yet mean it will work – for example, despite SQL having the capability to simulate Turing machines, we do not expect it to perform well with complex graph algorithms. Likewise here, the first subquery enumerates paths between two different nodes, and even the number of simple paths between two nodes in a graph can grow as fast as $O(n!/n^2)$.

We now show experimentally that queries of this kind have no chance to work even on very small graphs. A small obstacle is that there is not yet any available implementation of GQL, and Cypher, the closest language, chose not to have EXCEPT. However, there is a way around it in Cypher by using list functions that can detect the violation of the "value in edges increases" condition:

```
MATCH p=()-[*2..]->()
WITH p, reduce(acc=relationships(p)[0].val,
v in relationships(p) |
```



(a) $p = 0.1$ **(b)** $p = 0.2$ **(c)** $p = 0.3$

(d) $p = 0.4$ **(e)** $p = 0.5$

Figure 3: Timeouts and median running time of Neo4j for $Q^{\mathsf{E}}_{\uparrow}$

```
CASE  WHEN acc=-1 THEN -1
WHEN v.val>=acc THEN v.val
ELSE -1 END) AS inc
WHERE NOT inc = -1
RETURN p
```

We then tested its median running time in Neo4j[1], as well as the percentage of queries that time out (the timeout is set at 300 seconds). When more than 50% of queries time out, the median is shown as ∞ms. Otherwise the running time is computed as the median over 10 different graphs, with 1 run per graph (and an additional run 0 to generate the appropriate indices).

The graphs on which we tested the query are the random graphs $G(n, p)$ [11] on n nodes where an edge exists between two nodes with probability p. We considered values p between 0.1 and 0.5, with the step 0.1. As for data values in edges, they are also randomly generated, between 0 and 100. The reason we used this simple model of synthetic property graphs is that it very convincingly demonstrates that the above implementation of $Q^{\mathsf{E}}_{\uparrow}$ has no chance to work in practice. Even with the smallest probability of 0.1, with a mere 24 nodes timeout was observed in more than 50% of all cases, and with just 30 nodes in *all* cases. As the probability p increases (meaning that there are more edges in the graph, and thus the number of paths increases), the cutoff for everything-times-out dropped to fewer than 10 nodes!

We add two notes here. Queries $Q^{\mathsf{N}}_{\neq}$ and $Q^{\mathsf{E}}_{\neq}$ can be similarly expressed, with the bottleneck query matching all paths being identical to the above, hence resulting in a very similar behavior. But behavior is *not* observed with $Q^{\mathsf{N}}_{\uparrow}$ as it can be expressed directly with Cypher and GQL pattern matching, without having to find an exponential number of paths.

Other systems. Even though it appears that bypassing expressivity bounds by generating exponentially many queries should not have a chance to work (as our Neo4j tests confirm), we wanted to completely exclude the possibility that this behavior could be due to one specific implementation. Thus, to confirm the above results, we ran the same tests on two other systems: Memgraph, a graph-only database that uses Cypher as its query language, and DuckDB, a relational database that implements SQL/PGQ as an extension [39].

The results confirm that the problem is the way the query is written rather than a particular implementation. In fact, the limits of Neo4j and Memgraph are almost identical as shown in Fig. 4 which reports numbers of nodes at which 50% and 100% of runs timeout. However, it must be noted that for the configurations that do not timeout, the performance of Memgraph appears to be much more efficient than that of Neo4j, with almost all test cases taking less than 1 ms. This might be explained by the fact that the timing procedures are different for the two systems as Memgraph does not make query execution time available to the driver.

The performance of DuckDB was tested using the SQL/PGQ query below; it first finds the shortest paths in the graph using the PGQ pattern matching syntax (the inner query), then checks, in SQL, that the edge weights appear in sorted order (the outer query).

```
WITH q1 AS (SELECT *, unnest(path_edges) AS e_id
    FROM GRAPH_TABLE (testgraph
      MATCH
      p = ANY SHORTEST (n1:N)-[e:E]->{2,}(n2:N)
      COLUMNS (edges(p) AS path_edges) ),
    LATERAL (SELECT edges.weight,
                    edges.row_id FROM edges) )
SELECT path_edges, array_agg(weight) AS weights
FROM q1 WHERE e_id=row_id
GROUP BY path_edges
HAVING ARRAY_AGG(weight) =
       ARRAY_SORT(ARRAY_AGG(weight));
```

While the results for DuckDB appear to be better than for native graph systems (it can handle 164 nodes with the lowest probability $p = 0.1$ before running out of memory), there is a simple explanation for it: the difference in *path semantics*. The only path

[1]The testing program is written in Go and communicates with Neo4j (v5.18.1) via the Neo4j Go driver. All tests were executed on a machine with the following configuration: 16 Intel i7-10700 @ 2.90GHz CPUs, 16GB RAM, Ubuntu 22.04.3 LTS

	Neo4j			Memgraph			DuckDB		
	p=0.1	p=0.3	p=0.5	p=0.1	p=0.3	p=0.5	p=0.1	p=0.3	p=0.5
# nodes at 50% timeout	24	12	7	26	12	8	N/A	N/A	N/A
# nodes at 100% timeout / OOM	30	13	8	28	13	9	164	131	128

Figure 4: Performance comparison of Neo4j, Memgraph, and DuckDB

semantics available for repeated patterns in DuckDB is *shortest path*, whereas the only one available in Neo4j and Memgraph is *trail*, which matches paths that do not go through the same edge twice. In most graphs there are significantly more trails than shortest paths between any given nodes, hence the number of candidate paths to be checked is much higher for Neo4j and Memgraph than for DuckDB. Even with this, however, the way of bypassing expressivity bounds by generating a large number of paths can only handle very small graphs, not even reaching 200 nodes.

8 WHAT WE LEARNED ABOUT LANGUAGE DESIGN PROBLEMS

A common way of enhancing the capabilities of query languages is this: (1) an initial design is produced; (2) user-demanded queries that are not expressible in this initial design are identified; (3) their inexpressibility is formally confirmed; (4) language deficiencies that led to (3) are identified; (5) these language deficiencies are fixed. Note that (5) is often an iterative process that involves a deep analysis of possible language design enhancements. Returning again to the example of recursive queries in SQL, note that while it was quite clear that programmers cannot write queries such as transitive closure, it took a significant effort to confirm this formally, and subsequently hundreds of papers analyzed the expressiveness and complexity of datalog and fixed-point extensions of relational calculus, until the linear datalog approach was chosen as the right one to add to the SQL standard.

Where does this workflow put us with respect to the newly developed graph standards, SQL/PGQ and GQL? Prior to this paper we were at stage (2); the results of this paper complete step (3) and put us at the beginning of stage (4). Thus, in this last section we outline further developments as we envision them now: what language deficiencies cause limited expressiveness, and how they could be addressed.

The main limitation of current graph query languages is that they *lack compositionality*. They are essentially graphs-to-relations languages, as opposed to graph-to-graph languages (though some ad hoc functionalities exist for viewing relational outputs into graphs, such as the Neo4j browser or its data science library facilities [27] that can extract graphs from relations for analytics tasks; also the study of graph views is in its infancy [25]). The lack of compositionality also manifests itself in the information flow in graph query languages. It happens in one direction, from graphs to relations: patterns turn graphs into relations, but there are no natural ways of going in the other direction, and use results of relational operations of either RA or LCRA to create new graphs. Of course real-life languages with their extended functionalities provide such ways (e.g., via libraries as mentioned above, or by writing data and using it to create new graph views in PGQ) – but the main point is that these lie *outside* the realm of a declarative query language.

With the understanding of causes of limitations, the focus needs to be shifted to ways to remedy those, i.e., step (5) above. If the story of SQL is any indication, this will require a considerable research effort. Still, we can outline what we think are possible approaches to fixing the non-compositionality issues.

Pattern matching restricted to matched paths. While graph-to-graph languages are the ultimate long-term goal, in the short term some of non-compositionality can be fixed by letting pattern matching operate on previously matched patterns. We explain this idea by the query $Q^{\mathsf{E}}_{\uparrow}$. To find such paths from ℓ_1-labeled nodes to ℓ_2-labeled nodes, we can first match paths from ℓ_1-nodes to ℓ_2-nodes and then exclude those where the "increasing value in edges" condition is violated. Specifically, we start with the pattern $\psi := (x) \rightarrow^{1..\infty} (y)\langle \ell_1(x) \wedge \ell_2(y)\rangle$ and then create a new pattern

$$\psi \mid \neg\exists(\xrightarrow{u}\ ()\ \xrightarrow{v}\ \langle u.k \geq v.k\rangle) \tag{1}$$

The pipe operator | means that the pattern $\xrightarrow{u}\ ()\ \xrightarrow{v}\ \langle u.k \geq v.k\rangle$ is evaluated on the result of the match of pattern ψ, and the condition $\neg\exists$ says that there are no matches for it. This will ensure that in the match for ψ there are no consecutive edges for which the value of their property does not increase.

Constructing graph elements from tables. In current graph languages the flow of information is in the direction from graph to constructed relations. What if we could reverse it as well and construct new graph elements from relations? As an example of this, suppose we could construct new nodes with label *old_edge* whose ids are edge ids in a graph G, and a new edge with a lable *new_edge* which connects two such nodes coming from edges e_1 and e_2 of G if the target of e_1 is the source of e_2. Then

$$(x)\big((u) \xrightarrow{e} (v)\langle u.k < v.k \wedge \mathit{new\_edge}(e)\rangle\big)(y)$$

(i.e., $Q^{\mathsf{N}}_{\uparrow}$ on the constructed graph) expresses $Q^{\mathsf{E}}_{\uparrow}$ on G. This powerful idea was already present in an early theoretical language of [13] but never properly explored in the context of practical graph languages.

We conclude with a remark that language design is a delicate process in which the right balance must be struck between expressivity and complexity. It is not as simple as "add these features"; much new research is required as they come with complexity consequences. To give one example, in (1) replace $\xrightarrow{u}\ ()\ \xrightarrow{v}\ \langle u.k \geq v.k\rangle$ with $(u) \rightarrow^{1..\infty} (v)\langle u.k = v.k\rangle$. This results in query $Q^{\mathsf{N}}_{\neq}$ known to be NP-hard in data complexity [30]. Thus, such extensions are very sensitive to small syntactic changes. This however should simply be viewed as an invitation to start a research investigation into extensions of GQL and PGQ that allow desirable queries without unmanageable computational overhead. Once again, the past history of SQL tells us that such research could be very productive and have a great influence on the language.

ACKNOWLEDGMENTS

This research was supported by ANR Project VeriGraph, ANR-21-CE48-0015 (Leonid Libkin), a grant from RelationalAI to IRIF, Poland's National Science Centre grant 2018/30/E/ST6/00042 (Alexandra Rogova) and Israel Science Foundation 2355/24 (Liat Peterfreund).

REFERENCES

[1] S. Abiteboul and V. Vianu. Computing with first-order logic. *J. Comput. Syst. Sci.*, 50(2):309–335, 1995.

[2] S. Abriola, P. Barceló, D. Figueira, and S. Figueira. Bisimulations on data graphs. *J. Artif. Intell. Res.*, 61:171–213, 2018.

[3] F. N. Afrati, M. Gergatsoulis, and F. Toni. Linearisability on datalog programs. *Theor. Comput. Sci.*, 308(1-3):199–226, 2003.

[4] A. V. Aho and J. D. Ullman. The universality of data retrieval languages. In A. V. Aho, S. N. Zilles, and B. K. Rosen, editors, *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, USA, January 1979*, pages 110–120. ACM Press, 1979.

[5] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, and D. Vrgoc. Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5):68:1–68:40, 2017.

[6] M. Arenas, P. Barceló, L. Libkin, W. Martens, and A. Pieris. *Database Theory*. Open source at https://github.com/pdm-book/community, 2022.

[7] P. Barceló. Querying graph databases. In *Principles of Database Systems (PODS)*, pages 175–188, 2013.

[8] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4):31:1–31:46, 2012.

[9] B. R. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, S. Rondelli, A. Ryazanov, M. Schmidt, K. Sengupta, B. B. Thompson, D. Vaidya, and S. Wang. Amazon Neptune: Graph data management in the cloud. In M. van Erp, M. Atre, V. López, K. Srinivas, and C. Fortuna, editors, *Proceedings of the ISWC 2018 Posters & Demonstrations, Industry and Blue Sky Ideas Tracks co-located with 17th International Semantic Web Conference (ISWC 2018), Monterey, USA, October 8th - to - 12th, 2018*, volume 2180 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2018.

[10] M. Benedikt, A. W. Lin, and D. Yen. Revisiting the expressiveness landscape of data graph queries. *CoRR*, abs/2406.17871, 2024.

[11] B. Bollobás. *Random Graphs*, volume 73 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 2011.

[12] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000*, pages 176–185, 2000.

[13] M. P. Consens and A. O. Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 404–416. ACM Press, 1990.

[14] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In U. Dayal and I. L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, pages 323–330. ACM Press, 1987.

[15] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, H. Voigt, O. van Rest, D. Vrgoč, M. Wu, and F. Zemke. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD*, pages 1–12. ACM, 2022.

[16] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee. Aggregation support for modern graph analytics in tigergraph. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*, pages 377–392. ACM, 2020.

[17] A. Eisenberg and J. Melton. SQL: 1999, formerly known as SQL 3. *SIGMOD Rec.*, 28(1):131–138, 1999.

[18] R. Fagin. Monadic generalized spectra. *Math. Log. Q.*, 21(1):89–96, 1975.

[19] M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. A query language for a web-site management system. *SIGMOD Rec.*, 26(3):4–11, 1997.

[20] N. Francis, A. Gheerbrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, A. Rogova, and D. Vrgoc. GPC: A pattern calculus for property graphs. In F. Geerts, H. Q. Ngo, and S. Sintos, editors, *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 241–250. ACM, 2023.

[21] N. Francis, A. Gheerbrant, P. Guagliardo, L. Libkin, V. Marsault, W. Martens, F. Murlak, L. Peterfreund, A. Rogova, and D. Vrgoc. A researcher's digest of GQL. In F. Geerts and B. Vandevoort, editors, *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece*, volume 255 of *LIPIcs*, pages 1:1–1:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[22] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[23] H. Gaifman and M. Y. Vardi. A simple proof that connectivity of finite graphs is not first-order definable. *Bull. EATCS*, 26:43–44, 1985.

[24] S. Ginsburg and E. H. Spanier. Bounded Algol-like languages. *Transactions of the American Mathematical Society*, 113:333–368, 1964.

[25] S. Han and Z. G. Ives. Implementation strategies for views over property graphs. *Proc. ACM Manag. Data*, 2(3):146, 2024.

[26] P. Hell and J. Nešetřil. *Graphs and homomorphisms*. Oxford University Press, 2004.

[27] A. E. Hodler and M. Needham. Graph data science using neo4j. In D. A. Bader, editor, *Massive Graph Analytics*, pages 433–457. Chapman and Hall/CRC, 2022.

[28] N. Immerman. *Descriptive complexity*. Springer, 1999.

[29] L. Libkin. *Elements of Finite Model Theory*. Springer, 2004.

[30] L. Libkin, W. Martens, and D. Vrgoč. Querying graphs with data. *Journal of the ACM*, 63(2):14:1–14:53, 2016.

[31] T. Lindaaker. Predicates on sequences of edges. Technical report, ISO/IEC JTC1/ SC32 WG3:W26-027, 2023.

[32] W. Martens, M. Niewerth, and T. Popp. A trichotomy for regular trail queries. *Log. Methods Comput. Sci.*, 19(4), 2023.

[33] PRQL. *Pipelined Relational Query Language*, 2024. https://prql-lang.org.

[34] J. L. Reutter, M. Romero, and M. Y. Vardi. Regular queries on graph databases. *Theory Comput. Syst.*, 61(1):31–83, 2017.

[35] P. Selmer. Existing languages working group: GQL influence graph, 2019. https://www.gqlstandards.org/existing-languages.

[36] C. Sharma, R. Sinha, and K. Johnson. Practical and comprehensive formalisms for modelling contemporary graph query languages. *Inf. Syst.*, 102:101816, 2021.

[37] J. Shute, S. Bales, M. Brown, J. Browne, B. Dolphin, R. Kudtarkar, A. Litvinov, J. Ma, J. D. Morcos, M. Shen, D. Wilhite, X. Wu, and L. Yu. SQL has problems. we can fix them: Pipe syntax in SQL. *Proc. VLDB Endow.*, 17(12):4051–4063, 2024.

[38] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. T. Xie. Sqlgraph: An efficient relational-based property graph store. In T. K. Sellis, S. B. Davidson, and Z. G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1887–1901. ACM, 2015.

[39] D. ten Wolde, G. Szárnyas, and P. A. Boncz. Duckpgq: Bringing SQL/PGQ to duckdb. *Proc. VLDB Endow.*, 16(12):4034–4037, 2023.

[40] O. van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, pages 1–6, 2016.

[41] H. Wickham, R. François, L. Henry, K. Müller, and D. Vaughan. *dplyr: A Grammar of Data Manipulation*, 2023. R package version 1.1.4, https://github.com/tidyverse/dplyr.

[42] N. Yakovets, P. Godfrey, and J. Gryz. Evaluation of SPARQL property paths via recursive SQL. In L. Bravo and M. Lenzerini, editors, *Proceedings of the 7th Alberto Mendelzon International Workshop on Foundations of Data Management, Puebla/-Cholula, Mexico, May 21-23, 2013*, volume 1087 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2013.

[43] F. Zemke. For each segment discussion. Technical report, ISO/IEC JTC1/ SC32 WG3:BGI-022, 2024.