Dangers of List Processing in Querying Property Graphs

AMÉLIE GHEERBRANT, Université Paris Cité, CNRS, IRIF, France
LEONID LIBKIN, RelationalAI & University of Edinburgh, France & UK
ALEXANDRA ROGOVA, Université Paris Cité, CNRS, IRIF, France

The workhorse of property graph query languages such as Cypher and GQL is pattern matching. The result of pattern matching is a collection of paths and mappings of variables to graph elements. To increase expressiveness of post-processing of pattern matching results, languages such as Cypher introduce the capability of creating lists of nodes and edges from matched paths, and provide users with standard list processing tools such as `reduce`. We show that on the one hand, this makes it possible to capture useful classes of queries that pattern matching alone cannot do. On the other hand, we show that this opens backdoor to very high and unexpected expressiveness. In particular one can very easily express several classical NP-hard problems by simple queries that use `reduce`. This level of expressiveness appears to be beyond what query optimizers can handle, and indeed this is confirmed by an experimental evaluation, showing that such queries time out already on very small graphs. We conclude our analysis with a suggestion on the use of list processing in queries that while retaining its usefulness, avoids the above pitfalls and prevents highly intractable queries.

CCS Concepts: • **Information systems → Graph-based database models**.

Additional Key Words and Phrases: Graph databases; graph query languages; list processing; intractability; performance

ACM Reference Format:
Amélie Gheerbrant, Leonid Libkin, and Alexandra Rogova. 2025. Dangers of List Processing in Querying Property Graphs. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 144 (June 2025), 25 pages. https://doi.org/10.1145/3725281

1 Introduction

In the past decade and a half, graph databases have risen to a high level of prominence and are expected to play an ever increasing role in data management and analytics tasks [2, 38, 43]. Their appeal to the end user is due to the way they bridge the gap between the conceptual view of data and its representation in the database, where data is viewed just as it may have been depicted on a blackboard. The influence of graph databases is witnessed by a proliferation of products (Neo4j, Oracle, Amazon, SAP, Google, TigerGraph, etc) and strong predictions of their influence on data management tasks (such as estimates of graph data being used in up to 80% of data analytics tasks[1]). Notably, the International Organization for Standardization (ISO) has recently produced two new international standards for relation-based and native graph query languages. The first, known as SQL/PGQ, adds property graph querying to SQL (published by ISO in 2023 as part 16 of the SQL Standard) while the other, known as GQL, is a native graph query language (published in April 2024, also by ISO). These ISO standards, both coming from the same committee that produced and

[1]As per Gartner.

Authors' Contact Information: Amélie Gheerbrant, Université Paris Cité, CNRS, IRIF, Paris, France, amelie@irif.fr; Leonid Libkin, RelationalAI & University of Edinburgh, Paris & Edinburgh, France & UK, l@libk.in; Alexandra Rogova, Université Paris Cité, CNRS, IRIF, Paris, France, rogova@irif.fr.

Please use nonacm option or ACM Engage class to enable CC licenses
This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2836-6573/2025/6-ART144
https://doi.org/10.1145/3725281

maintains the SQL standard, apply specifically to the model of *property graphs*. These are graphs in which both nodes and edges may possess labels as well as zero or more properties. For example, a graph may have two nodes labeled `Person` connected by an edge labeled `Friend`, where `Person` nodes have properties `name` of type `string` and `age` of type `int` whereas the `Friend` edge has a property `since` of type `date`.

While many industrial and academic contributions influenced the design of these languages [41], by far the most prominent of them was Cypher [19], developed at Neo4j, and implemented by multiple other systems. It is still the most popular real-life graph query language today[2]. But its original design comes with known limitations, which are addressed in various ways in the design of GQL and SQL/PGQ and in the enhancement of Cypher itself. One example of such a limitation is the weakness of Cypher's patterns: they cannot express all regular path queries (RPQs) [11]. There are two ways of fixing such expressivity gaps. One is to add a specialized construct designed for a specific task. The other is to add a general programming construct that significantly elevates the power of the language.

Let us analyze how these two approaches worked in the context of graph query languages, in particular, in the light of the new standardization efforts. GQL and SQL/PGQ looked at Cypher's perceived inabilities to express certain queries, most of all extensively studied RPQs, conjunctive RPQs (CRPQs [12]) and regular queries [36]. In response to this, the key construct of regular queries were added, namely the ability to repeat any path pattern an arbitrary number of times. For example, while Cypher allows patterns such as

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

that look for `a`-labeled paths of arbitrary length between `x` and `y`, GQL and SQL/PGQ allow patterns such as

```
(x) ( -[e1:a]-> -[e2:b]-> WHERE e1.p1 < e2.p2)* (y)
```

that match paths labeled `ab...ab` in which the property `p1` of every `a`-labeled edge is less than the property `p2` of the `b`-labeled edge that follows it. Since Cypher itself is not a standard, it is up to individual vendors to decide whether to adapt to such changes. Some do (e.g., Neo4j in their latest versions), but most do not (at least not yet).

But these are not the only desirable yet unavailable queries. In general, analyzing or aggregating properties that occur along a matched path is beyond both Cypher and GQL abilities. For example, checking if properties of edges follow a specific pattern (say, they increase, or are all different) is not something one can easily do in pattern matching alone. Neither is aggregating over values of properties (e.g., adding up values in edges or nodes along the path), nor finding paths in which a *negation* of an otherwise expressible property holds (path languages are in general not closed under negation as doing so has significant complexity repercussions [26]).

To add such queries, Cypher opted to resort to a very general programming construct: `reduce` over lists. In general, the language lets the programmer construct lists, for instance of nodes and edges that occur on a path, and then use `reduce` (also known as `fold`), a basic list processing routine, over them. As this significantly enhances expressivity, the question we would like to ask is:

> *What are the consequences of adding lists and general list processing constructs to a graph query language?*

There are two main reasons for asking this question. First, the existing practice of query language design tells us that such additions can have a significant effect on the power and complexity of the language and consequently on the ability to optimize queries. As an example, consider SQL. Recursion was added in the SQL 1999 standard, to overcome some of its limitations; however

[2]As per the DB Engines ranking.

together with already existing aggregation it made the language very powerful: Turing-complete, to be precise. This of course impairs the ability to optimize queries.

The second reason is directly related to the development of new graph query languages. GQL, released less than a year ago, is in its first edition. Its current state is similar to the first version of the SQL standard in 1986, which laid the foundation for many additions in subsequent versions of the standard. With the next GQL release most likely around 2029, it is expected that there will be additions to address its expressivity gaps. As explained previously, in Cypher, the strongest influence on GQL, these gaps are addressed by means of adding list processing facilities. However, these have not been studied from the point of view of their expressive power nor performance. Thus, such a study is necessary to better inform decision making regarding new features that inevitably will be added to future versions of GQL. In fact we have a good precedent: the study of the complexity of property paths in SPARQL [5, 28] prevented the standard from adopting an unworkable semantics.

We next outline our contributions.

At first glance the addition of list processing to Cypher looks like a reasonable solution and a welcome increase in the expressiveness of the language, that uses tools familiar to many programmers. Indeed, we show how this gives us the power of RPQs [11] and of many extensions such as extended CRPQs [7], which add path comparisons, and data CRPQs [26], which add comparisons of property values along paths.

However, the increase in expressiveness is too much for a query language. To start with, nested `reduce` can simulate the powerset operator. Clearly, we want to avoid this extreme increase of expressiveness, and thus we restrict the uses of `reduce` as follows:

- they do not produce intermediate results larger than the lists they operate on; and
- their final result is a single value.

But it turns out that even this much simplified use of list operations lets us express several NP-complete problems such as Hamiltonian path or subset sum.

We show experimentally that such queries perform very poorly. We tested them on random graphs, and for edge probability > 0.2, they time out already on graphs with 10 nodes! While this may sound dramatic, recall that the number of simple paths in a graph grows exponentially with the number of nodes, reaching in the worst case $(N-2)!$. Query plans that use lists, especially for filtering out matched paths, are bound therefore to generate a huge number of possible paths. In fact, as one would expect from bounds on the number of paths, the performance degrades as graphs become less sparse (which incerases the number of paths to analyze). Furthermore, the size of graphs on which we see timeouts is so small that we could only reasonably do experiments on synthetic data, as real data with such super-small graphs is hardly in existence.

One could argue though that this is not a "big deal". Cypher queries use, by default, *trail semantics*: a matched path cannot go twice over the same edge. GQL and SQL/PGQ add other modes of path pattern matching: simple paths (cannot go twice over the same vertice), shortest paths, or arbitrary paths if the pattern can only be matched by paths of fixed length [13]. Of these, matching trail and simple path is already NP-hard [19, 31, 32].

Thus, to understand whether list processing or trail semantics is responsible for the bad performance, we conduct two additional sets of experiments. The first looks at theoretically intractable trail pattern matching, and the second on using lists with a tractable path matching mode, namely shortest paths.

Regarding the first, notice that matching trail paths did not stop Cypher from being a dominant graph query language. This is due to the fact that most instances of trail pattern matching are actually tractable [10, 31]. A notable case where it is NP-complete is matching a trail path whose

labels form a word in A^*BA^*. However, in this case we notice that the sizes of graphs that can be handled are several times bigger than those on which the Hamiltonian path query timed out. Moreover, running times are significantly lower.

To further discount the hypothesis that the trail semantics is the main culprit, we look at the subset sum query, which uses lists but relies on shortest paths, rather than trails. With this query, the results are in line with those for Hamiltonian path rather than A^*BA^* trails: queries time out on very small graphs. Hence, it is the use of lists, rather than the trail semantics, that results in a very poor performance.

Another question is whether queries that exhibit such behaviour are actually realistic. After all, as already mentioned, with recursion and aggregation SQL has sufficient power to express all computable queries, but in practice programmers do not write Turing machine simulators in SQL – with few exceptions. We provide evidence that offending queries are rather natural. To start with, multiple books, blogs, and manuals for programmers advertise precisely the kind of queries that create problems (we shall analyze their syntactic shape later). This by itself is not too surprising since lists with powerful recursion over them are the only recourse programmers have to achieve the desired expressiveness. Furthermore, we ran a user study that was designed to answer two questions: (a) can users easily write very high-complexity queries? and (b) do they estimate how bad complexity consequences are? With the clear positive answer to the first and negative to the second question, we confirmed that the problem can occur in practice.

Our last technical question is whether SQL, especially with recursive common table expressions, would perform better than a native graph database such as Neo4j on these problematic queries. Recursive queries give SQL enough power to express them. We observe that Cypher handles the theoretically NP-hard query of matching A^*BA^* trail paths much better than SQL. However, for queries where Cypher requires list processing – Hamiltonian path and subset sum – SQL performs slightly better than Cypher, but still cannot handle even modest sized graphs.

These results are a consequences of language design, namely putting list operations in a graph query language, rather than the fault of a particular implementation. Indeed, in the absence of useful optimizations that can be applied, they force the engine to build a very large number of paths, rendering queries completely impractical. This work was done in the context of very active work on the next version of GQL. Since it extends the expressivity of the language, it is tempting to adapt Cypher's solution. However, our results show that it should not be adapted as-is. In terms of changes required, our results point to specific restrictions that need to be imposed on the use of lists. We dedicate a section to it, but here can summarize them rather simply:

- It is ok to use lists to post-process results of pattern matching as long as nested lists are not created;
- It is not, however, advisable, to use expressions with list operations to *filter* the set of selected paths.

These simple recommendations have the advantage of being easily adopted by query languages, by imposing syntactic restrictions on where and how list operations can occur.

Organization. We give a brief overview of the main operations of Cypher in Section 2. Then in Section 3 we show some shortcomings of Cypher, which motivated the introduction of lists. In Section 4 we show how lists help express many useful queries, particularly RPQs and their extensions. Section 5 demonstrates that the same list facilities lead to problems, namely expressing computationally intractable queries. In Section 6 we provide an experimental evaluation of such highly intractable queries, and show that it is list operations, rather than the trail semantics, that are responsible for the poor performance. Results of the user study are presented in Section 7. In Section 8 we compare a native graph implementation with SQL. Finally, in Section 9 we discuss

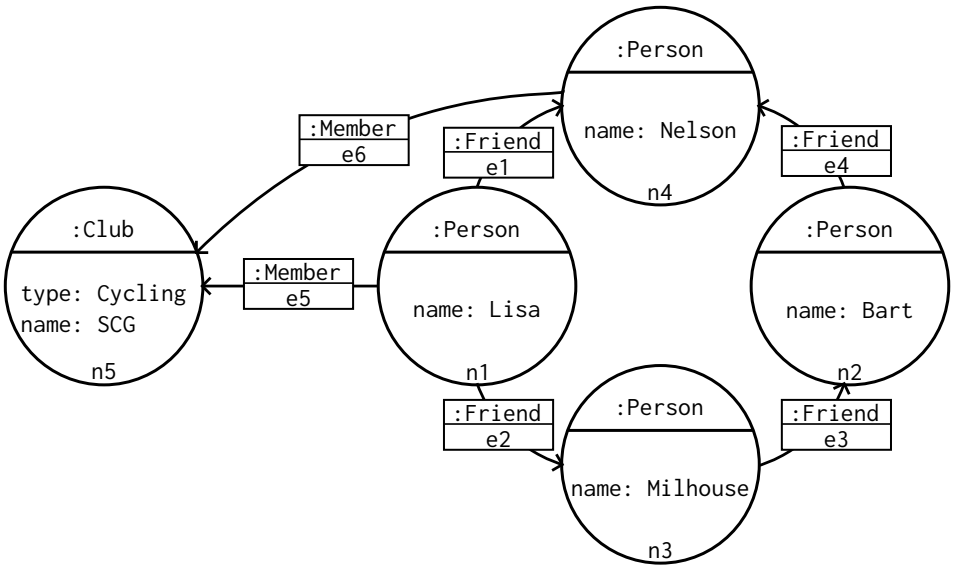


Fig. 1. A cycling property graph

what it means for language design, especially for future enhancements to GQL. Supplementary material (full code and results) can be found at this at this link.

2 Cypher: a quick introduction

Cypher is currently the most commonly used query language for property graphs (as defined in [2, 43]). The main difference between property graphs and the usual oriented graphs structures is that vertices and edges can carry data in the form of multiple key-value pairs (such as `name:"Lisa"`). Nodes and edges can additionally carry an arbitrary number of labels (such as `Person` or `Club`), though some systems restrict edges to at most one single label, without putting restrictions on node labels. Both nodes and edges are stored using some unique system dependent identifier.

Figure 1 represents a property graph with five nodes, four of them labeled `Person` and having attribute `name` and one labeled `Club` with an attribute `type` equal to `Cycling` and an attribute `name` equal to `Springfield Cycle Gang`. The graph also contains six edges, all oriented, four with label `Friend` and two with label `Member`. Each node and edge also carries a unique identifier (`n1, e1,` and so on).

Since Cypher, unlike GQL, lacks a standard, and implementations can differ rather significantly, when we refer to Cypher we mean its description in the standard academic reference [19]. Some implementations, such as Neo4j, have moved from this description into the direction of GQL quite significantly, though others, e.g., Memgraph and Amazon, are much closer to the original Cypher. Cypher queries take as input a graph and usually end with a `RETURN` statement, outputting a table. A query is composed of statements, executed in sequences; this is referred to as linear composition [19, 43]. Each statement in turn takes a graph and a table (called driving or working table), and returns a graph and a potentially modified working table. For read-only queries, which are our main concern here, statements do not change the input graph.

An example is the following query, which returns the names of friends of Milhouse who have at least one but no more than three other friends who belong to a cycling club :

```
MATCH (p1:Person)-[:Friend]-(p2:Person)-[:Friend]-(p3:Person),
      (p3)-[:Member]->(c:Club)
WHERE p1.name='Milhouse' AND c.type='Cycling'
WITH p2, COUNT(DISTINCT p3) AS fof
WHERE fof <= 3 AND fof >= 1
RETURN p2.name
```

The core of the Cypher language is its pattern matching mechanism. Basic building blocks are patterns, which have a user friendly "ASCII art" flavor, as witnessed by the two patterns in the `MATCH` clause above. The first one connects three nodes labeled `Person` using two `Friend` relationships. The second pattern, starting on the second line and separated from the first by the comma, is joined to the first one via the third `Person` node using the variable `p3` and connects it to a node labeled `Club` via a `Member` relationship. Variables `p1`,`p2`,`p3` and `c` are used to bind matches in the working table. The execution of a Cypher query starts with an input graph and a table containing one empty tuple `()` to which pattern matching adds tuples with bindings for `p1, p2, p3` and `c`. These tuples are then filtered by the `WHERE` clause (checking `p1`'s name and `c`'s type), resulting in the following table.

p1	p2	p3	c
n3	n2	n4	n5
n3	n1	n4	n5

The subsequent `WITH` clause modifies this table, by retaining only matches for `p2`, and computing, for each of the values of `p2`, the number of distinct values of `p3` that occur with it in the tuple (and binding it to attribute `fof`).

This results in the modified driving table shown below.

p2	fof
n2	1
n1	1

Finally, the `RETURN` clause acts similarly to SQL's `SELECT`, forming the output; in our example the value of the attribute `name` of the node matched to `p2` will be output, which corresponds to Bart and Lisa for the graph in Figure 1.

Notice that the orientation of both `Friend` edge patterns is not specified in the query and so the path can traverse them in *any direction*, thus creating a first solution traversing the nodes corresponding to Milhouse then Bart then Nelson, and a second solution going from Milhouse to Lisa to Nelson. As `p3` is Nelson in both cases, the value of `fof` remains 1.

An important aspect of Cypher pattern matching is the enforced *trail semantics*, meaning each edge can be traversed at most once per path. Thanks to this rule, finiteness of matches is always guaranteed. For example if the condition on `p1` were to be changed from `p1.name='Milhouse'` to `p1.name='Lisa'`, there would be no valid answer as the only `Person` at distance 2 from Lisa who is also a Member of a cycling club is Lisa herself and such a path would necessarily use the same edge (between either Lisa and Milhouse or Lisa and Nelson) twice, even if in opposite directions.

It is a crucial feature of any reasonable graph query language to allow the retrieving of paths of arbitrary length (which falls short of the expressive power of usual conjunctive queries). Cypher enables this via a restricted form of Kleene star. This is implemented by the use of variable length relationships inside patterns, indicating that some possibly unbounded number of relationships should be traversed. The most basic form of such patterns is `()-[*]-()`, which can be matched to any path (once again with no repeated edges). Minimal and maximal length of paths to be matched can be set, as well as admissible labels for edges, as in `()-[:l1|l2*2..3]-()`, where matches will

be restricted to paths of length 2 to 3 where edges are only labeled with `l1` or `l2`. However, it is not possible to directly label the path with a regular expression.

This sets Cypher , as defined in [19], apart from regular path queries [11], the main graph query language considered in the research literature, and also from languages such as GSQL [14], PGQL [42], G-Core [1], and GQL [13, 18].

3 Limitations of Cypher: lists to the rescue

3.1 Cypher limitations

Cypher has a number of limitations. To start with, as originally designed, it cannot express all RPQs. Recall that an RPQ is given by a regular expression e over the alphabet of edge labels. Such a query returns pairs of nodes connected by a path whose edge labels form a word in the language of e. Cypher can express RPQs such as a^* by `(x)-[:a*]->(y)`; it can also express some more complex expressions such as a^*b^* by combining patterns such as `(x)-[:a*]->()-[:b*]->(y)`. However the main limitation of Cypher patterns is that the Kleene star $*$ can only be applied to (disjunctions of) edge labels, and not to more complex regular expressions. This renders even simple regular expressions such as $(aa)^*$ inexpressible with Cypher 's basic pattern matching mechanism. This limitation led to more expressive GQL pattern matching, partly adopted by latest versions of Neo4j but not yet others.

Similarly, more complex path queries are not definable with Cypher's basic pattern matching. These include (a) CRPQs [12], or joins of RPQs, (b) ECRPQ [7], which allow path comparisons with regular predicates (such as: lengths of paths p_1 and p_2 are the same, or the label of p_1 is a prefix of the label of p_2), (c) various extensions of CRPQs with handling data, such as checking for equality of property values in nodes or edges of matched paths [26].

There are other rather natural conditions on paths that cannot be expressed, neither in the original Cypher nor in GQL and SQL/PGQ. Recall that a path in a property graph is an alternating sequence of nodes and edges that starts and ends in a node [19], i.e., a sequence $p = n_1\ e_1\ n_2\ e_2 \ldots e_{m-1}\ n_m$ where each n_i is a node and e_j is an edge connecting n_j and n_{j+1} (meaning it either goes from n_j to n_{j+1} or from n_{j+1} to n_j or is an undirected edge between them). Assume now that each node has a property k and each edge has a property s. Consider the following properties of path p:

(1) Values in nodes increase: $n_1.k < n_2.k < \ldots < n_m.k$;
(2) Values in edges increase: $e_1.s < e_2.s < \ldots < e_{m-1}.s$;
(3) Values in nodes are different: $n_i.k \neq n_j.k$ for $1 \leq i < j \leq m$;
(4) Values in edges are different: $e_i.s \neq e_j.s$ for $1 \leq i < j < m$;
(5) Values in all nodes/edges are similar: $|n_i.k - n_j.k| < t$ for $1 \leq i < j \leq m$ and some threshold t, and likewise for edges.

Of these, only the first one can be expressed by a simple pattern (available in GQL and the latest version of Cypher):

```
MATCH (x) ((n1)->(n2) WHERE n1.k<n2.k)+ (y) RETURN x, y
```

Others cannot be expressed in pattern matching only; while intuitively clear, this was proved formally in [22] for items (2)–(4); item (5) follows by setting $t = 1$ for integer values of property k.

While these fairly simple properties are not expressible by patterns alone, there is a seemingly natural way to add them to the language. This was followed by Cypher that introduced the capability to define two *lists* for a path $\mathsf{p} = n_1\ e_1\ n_2\ \ldots e_{m-1}\ n_m$:

- `nodes(p)` $= [n_1, \ldots, n_m]$ of all nodes of `p`, and
- `relationships(p)` $= [e_1, \ldots, e_{n-1}]$ of all edges of `p`,

both in the order in which they appear in `p`.

Then the above queries are easy with some standard list functions. Checking that p conforms to $(aa)^*$ we need to check that every label is a (by `-[:a*]->`) and that `length(nodes(p))` is odd (or `length(relationships(p))` is even). For other conditions, we apply the standard `reduce` (or `fold`) function on lists that accumulates a value as it iterates over list elements:

$$\texttt{reduce}[\iota, f]([a_1, \ldots, a_n] = f\big(\ldots f(f(\iota, a_1), a_2), \ldots, a_n\big)$$

For example, to check whether the list of non-negative elements $[a_1, \ldots, a_n]$ is in the increasing order, we use $\iota = (0, \text{true})$ and $f\big((a, \text{truth\_value}), b\big) = \big(b, \text{truth\_value} \wedge (a.k < b.k)\big)$

3.2 Cypher support for lists

We now briefly outline Cypher operators for list manipulation.

Creating lists. There are several ways to generate a list. We already saw two, namely `nodes(p)` and `relationships(p)` for creating lists of nodes and edges of a path p. Entries of such lists are node and edge ids, and thus their labels and properties can be retrieved too. For a single graph element x (node or edge), `keys(x)` is the list of its property names, and for a node n, its list of labels is returned by `labels(n)`.

There are ways to create lists independently of graph elements: `range(i, j [, step])` returns the list containing all elements between `i` and `j`, where the difference between two consecutive elements is given by the value of the expression `step`. Also an arbitrary set `S` can be turned into a list by `collect(S)`. Here a set is the collection of elements matched by a pattern matching variable. For example, in the pattern `MATCH (n :Person) collect(n)`, the set corresponding to `n` would be all the nodes in the graph with label `Person` and `collect(n)` would return the list containing those nodes in arbitrary order.

Operations on lists. Alongside list creation functions, Cypher offers many ways of manipulating lists. The basic operations are:

- `L1 + L2` is the concatenation of `L1` and `L2`;
- `e IN L` checks if element `e` belongs to the list `L`;
- `L[n]` returns the element at position `n` in `L`.

The workhorse of list processing is the `reduce()` function (sometimes called `fold` in the context of functional programming). In its most general form it is given as

```
reduce(acc = init, x IN L | f(x, acc))
```

where `L` is a list, `init` is the initial value of the accumulator variable `acc` and `f(x,acc)` is a function applied to a list element `x` and the accumulator value `acc` to produce a new accumulator value:

```
reduce(acc=init, [] | f) = init
reduce(acc=init, x::L | f) = f(x, reduce(acc=init, L | f))
```

There are important special cases of `reduce` that have their own syntactic construct due to their frequent use. These are

- `all(L, p)` : it returns true if all elements of `L` satisfy the predicate `p` (in this case `init=true` and `f` is conjunction);
- `none(L, p)` checks if no element of `L` satisfies `p`; this is `all` applied to the negation of `p`.
- `any(L, p)` returns true if some element of `L` satisfies `p` (here `init=false` and `f` is disjunction);
- `size(L)` outputs the size of `L`; here `init=0` and `f` increments `acc` by 1.

Of course Cypher provides many other functions such as `head`, `tail`, `reverse`, `isEmpty`, as well as `UNWIND L` which creates a row for each element of `L`, but the above will suffice for our examples.

We shall consider two variants of using lists in Cypher: (1) with the full power `reduce`; and (2) Without the general `reduce` but with derived functions `all`, `none`, `any`, and `size`. We shall see that even the very simple fragment (2) is very expressive but already comes loaded with issues.

4 Lists are good

We now show how adding lists lets us express what was previously inexpressible in Cypher: RPQs, several of their extensions, and conditions on paths mentioned in the previous section.

Aggregation. A common use of lists is not a limitation of pattern matching per se, but aggregation. There are two forms of aggregation in graph languages: vertical and horizontal. The former is the usual relational aggregation over working tables.

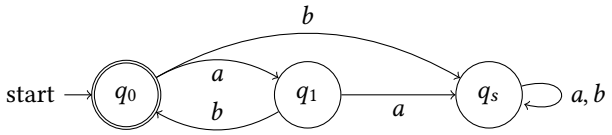
The latter works on a single tuple of a working table. Recall that such a tuple may have paths of arbitrary length as entries, and horizontal aggregates computes aggregate values over properties of nodes or edges of such paths. For example, the query below computes the cost of the different routes from Springfield to Shelbyville, as a weighted sum of total length (to account for the cost of gas) and tolls. These sums themselves are computed by `reduce` over the list `relationship(p)` of edges of paths `p`.

```
MATCH p=({name: Springfield}}-[*]->({name: Shelbyville})
WITH (r IN relationships(p) | r.length) AS lengths,
     (r IN relationships(p) | r.toll) AS tolls
RETURN reduce(sum=0, l IN lengths | sum+l)*1.8 +
       reduce(sum=0, t IN tolls | sum+t) AS cost
```

RPQs and CRPQs. Recall that an RPQ is given by a regular expression e over edge labels, and returns pairs of nodes connected by a path whose labels form a word in the language denoted by e. To see how to express this with lists in Cypher, we convert e into an equivalent deterministic finite automaton $\mathcal{A} = (Q, q_0, F, \delta)$ where $Q = \{q_0, q_1, \ldots, q_n\}$ is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of accepting states, and $\delta : Q \times \Sigma \to Q$ is the transition function. If $w = a_1 \ldots a_m$ is a string in Σ^*, it is accepted by $\mathcal{A}$ if there exists a sequence of states $r_0, r_1, \ldots, r_m$ with $r_0 = q_0$ and $r_n \in F$ such that $r_{i+1} = \delta(r_i, a_{i+1})$ for all $0 \leq i < m$. Therefore, it suffices to write a query that will simulate the run of the underlying automaton of an RPQ over the word composed of the edge-labels of the path.

We illustrate this by an example of a query inexpressible in the basic Cypher of [19], given by the regular expressions $(ab)^*$; the construction will make it clear how to extend it to all RPQs.

The automaton $\mathcal{A}$ has three states $\{q_0, q_1, q_s\}$, with q_0 both the initial and the final state; its transitions δ are shown below:



To emulate $\mathcal{A}$ as a Cypher query, we start by matching an arbitrary path p starting in a node labeled `Start` and then creating its list of edge labels, which we call `types_p`:

```
MATCH p = (:Start)-[*]-()
WITH [r IN relationships(p) | type(r)] AS types_p, p
```

We then emulate the run of $\mathcal{A}$ over `types_p` using `reduce`. The current state is stored in the accumulator variable `state` and the transition function is given as a list of `CASE` statements (one for each state) each containing a sub-list of `CASE` statements (one for each transition from that state) that returns the next state.

```
WITH reduce (state = 'q0', label IN types_p |
CASE state
    WHEN 'q0' THEN
        CASE label
            WHEN 'a' THEN 'q1'
            ELSE 'qs'
        END
    WHEN 'q1' THEN
        CASE label
            WHEN 'b' THEN 'q0'
            ELSE 'qs'
        END
    WHEN 'qs' THEN 'qs'
        END
END) AS final_state, p
```

We finally check whether the value returned by `reduce`, called `final_state`, is a final state (in our case, q_0) and return the path `p`.

```
WHERE final_state IN ['q0']
RETURN p
```

This approach can clearly be generalized to any finite automaton, by writing out explicitly the whole transition function δ as a series of `CASE` statements in the `reduce` combining function.

The same approach works for CRPQs which are joins of RPQs. Specifically a CRPQ [12] is a query of the form

$$Q(\bar{z}) \;:-\; x_1 \xrightarrow{e_1} y_1, \cdots, x_k \xrightarrow{e_k} y_k$$

where, for each $1 \leq i \leq k$, the query $x_i \xrightarrow{e_i} y_i$ is an RPQ given by the regular expression e_i, and $\bar{z}$ is a tuple of variables among x_i, y_i for $1 \leq i \leq k$. Such a query computes all the RPQs $x_i \xrightarrow{e_i} y_i$, joins them, and projects out variables in $\bar{z}$.

Again, we convert each e_i into a DFA $\mathcal{A}_i$ and follow the approach above, instead matching k paths:

```
MATCH p_1 = (x1)-[*]->(y1), ..., (p_k) = (xk)-[*]->(yk)
WITH [r IN relationships(p_1) | type(r)] AS types_p_1, p_1
....
WITH [r IN relationships(p_k) | type(r)] AS types_p_k, p_k
```

followed by k `reduce` statements simulating the automata $\mathcal{A}_1, \ldots, \mathcal{A}_k$, and then checking that all of them are in their respective final states, just as we did above for a single RPQ. Notice that the implicit join on variables of the same name is preserved in the translation. Finally this query concludes with a `RETURN` statement projecting out variables corresponding to $\bar{z}$.

Extended CRPQs. Using the same ideas, this translation can be adapted to Extended CRPQs (ECRPQs, as defined in [7]), which, as the name implies, extend CRPQs in two ways:

- They add the ability to talk about whole paths instead of just their endpoints. That is, they allow RPQs of the form $\pi = x \xrightarrow{e} y$, with π bound to paths that match the RPQ; and
- they can express conditions on multiple paths, specifically conditions specified by *regular relations*.

Examples of regular relations are path equality (p_1 and p_2 are labeled by the same word), prefix (the label of p_1 is a prefix of the label of p_2) and synchronous transformations (when letter a appears on p_1, letter b must appear in the same place on p_2). In the context of graph data, they arise in applications such as the Semantic Web, handling biological sequences, or route-finding see [4, 8, 23].

They have an associated automaton model, a synchronous multitape automaton, cf. [39]. With regular relations, one can test non-regular and even non-context-free properties of paths. For example,

$$Q(x,y) \quad :- \quad \pi_1 = x \xrightarrow{a^*} u,\ \pi_2 = u \xrightarrow{b^*} v,\ \pi_3 = v \xrightarrow{c^*} y,\\ \mathsf{el}(\pi_1,\pi_2),\ \mathsf{el}(\pi_2,\pi_3)$$

where el is the equal-length (regular) predicate, recognizes a path labeled by a word in the language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ which is neither regular nor context-free.

Despite the much increased expressive power, we can adapt the automa-based technique to ECRPQs by creating an additional structure that contains the list of edge-labels of all paths at any given point. This can be achieved by first matching m paths mentioned in the query, extracting their labels into m lists as before and then building a fresh array `path_labels`, of length equal to the longest path, such that `path_label`(i) contains the labels of all edges at position i in the matched paths (or a special symbol $\perp$ if such an edge does not exist). The transition function translation then follows the same structure as above, with a `CASE` statement for each combination of automaton state and permutation of letters from the edge label alphabet. As an example, to check if one path is a prefix of another, we simulate the following transition function:

```
reduce(state='q0', labels IN path_labels |
CASE state
    WHEN 'q0' THEN
        CASE WHEN labels[0]=labels[1] OR labels[0]=⊥
            THEN 'q0'
            ELSE 'qs'
        END
    WHEN 'qs' THEN 'qs'
END) AS final_state
WHERE final_state = 'q0'
```

Here q_0 is both initial and final, and q_s is the non-accepting sink state. As long as the label of the second path, stored in `labels[1]`, equals that of the first, stored in `labels[0]`, or the first path has ended, the automaton stays in q_0; otherwise it switches to and remains in q_s. Then reaching q_0 at the end indicates that the first path is a prefix of the second.

Queries comparing values in nodes and edges. To illustrate how value-based queries inexpressible in pattern matching can be expressed using lists, we give as an example the query "values in all edges from Start to End are different".

```
MATCH p=(:Start)-[*]->(:End)
WITH [r IN relationships(p) | r.val] AS values, p
WITH reduce(res=[true,[]], val IN values |
        CASE res[0]
            WHEN true THEN
                CASE WHEN val IN res[1] THEN [false,[]]
                     ELSE [true, res[1]+val]
                END
            ELSE res
        END
    ) AS result,p
WHERE result[0]=true
RETURN p
```

The query iterates over the edges of a path `p`, storing the values `r.val` instead of the labels. The `reduce` function checks the condition by storing all values encountered thus far in the second

element of the accumulator. If the next value is already present in the list, the first element of the accumulator is set to `false` and will stay `false` until the end of the computation, otherwise the second element remains `true` and the new value is added to the list. Notice that the accumulator of this `reduce` function is a complex object: it is a list of size 2, whose first element is a boolean and whose second element is a list of values of arbitrary (albeit all the same) type.

5 Lists are bad

As shown in section 4, the `reduce` function is a powerful tool as it gives a way to express conditions that go beyond regular expressions on any structure. In this section, we show how this expressive power can lead to intractable queries.

To start with, the high expressive power of `reduce` in the context of a query language is not surprising per se. Even for bags, that drop the order from lists, adding `reduce` and nesting allows queries whose complexity is a fixed-height tower of exponentials (e.g., k-EXPTIME complexity for any fixed k) and the class of encoded numerical functions is Kalmar-elementary [24, 27]. The latter optimistic name dates back to the early days of recursion theory where it meant "less than primitive recursive"; in reality it captures definitions given by second-, third, . . ., k-order logic, and is thus completely impractical.

To see where this extremely high complexity comes from, and crucially how to exclude an easy way of writing effectively non-computable queries, we note that Cypher imposes no restrictions on the accumulator value, nor the combining function. Hence a query computing the powerset of a set, such as the one below, which uses a list of lists as the accumulator and a second `reduce` as the combining function, is allowed.

```
WITH reduce(res=[[]], i IN range(n,m) |
    reduce(subres=[], j IN res |
        subres+[j+[i]]+[j]
    )
) AS powerset
RETURN powerset
```

The input in this example is a range of numbers (but could be any list). It uses two nested `reduce` functions. The outer loop iterates over a given set S and returns a set of subsets containing elements of S. The inner loop iterates over the current subsets and executes two operations: (1) it adds the current element to each subset, and (2) it creates a new subset that contains only the current element. As the main property of powersets is their exponential size, this query generates exponentially many (in the size of the list) results and is therefore unreasonably slow for any list except very small ones.

We clearly want to exclude such a behavior, so we consider several restrictions on `reduce`.

Restriction 1: no composite type accumulators. Since the powerset query relies heavily on accumulating lists in lists, the first restriction we consider is to disallow composite type accumulators (such as lists and maps). However, even under such a restriction it is easy to write intractable queries, in fact even avoiding the default trail semantics of Cypher and using a common shortest path semantics.

The problem we consider is subset sum: given a (multi)set S and a target-sum T, is there a subset S' of S whose elements sum up to T, i.e., $\sum_{s \in S'} = T$. It is known to be NP-hard [20] assuming a binary encoding for the integers (and is tractable under the unary encoding [15]).

The query computing the subset sum problem shows that this restriction is not sufficient.

Given a set S, we encode it with a graph that has $|S| + 1$ nodes and $2|S|$ edges defined as follows. Assume some enumeration $s_1, \ldots, s_k$ of elements of S. The graph has nodes $n_0, n_1, \ldots, n_k$, with n_0

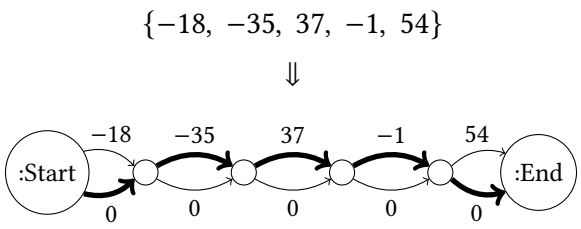


Fig. 2. Subset sum as a graph problem, in bold a solution for $T = 1$

having label *Start* and n_k having label *End*. All edges have the same label *Edge* and one property *value*. We have two edges from n_{i-1} to n_i, for $0 < i \leq k$, one with *value* $= 0$ and the other with *value* $= s_i$. See figure **??** for an illustration.

It is clear that all shortest paths from n_0 to n_k have the same length, there are 2^k of them, and along each path one chooses an edge from n_{i-1} to n_i that either has value 0, thereby skipping s_i from the sum, or value s_i, thereby adding it. Using flat lists that only have edges from the path following by *summing up* their elements we encode the subset sum problem by the following query.

```
MATCH p = allShortestPaths((:Start)-[:Edge*]->(:End))
WITH [r IN relationships(p) | r.value] AS values, p
WHERE reduce(sum = 0, v IN values | sum|+v) = $T
RETURN p
```

For the above shaped graphs, this query solves the subset sum problem by finding a path along this graph such that the sum of the edge values is equal to T.

Therefore, restricting `reduce` accumulators to primitive types only is not sufficient. We thus look at much more drastic restriction of `reduce` but even with that, we can still encode computationally intractable problems.

Restriction 2: the only permitted instances of `reduce` *are* `all` *and* `size`. In other words, only the four simplest incarnations of `reduce` from Section 3 are allowed. Notice that `none` and `any` are expressible with `all` using negation. This may seem to be a draconian restriction, eliminating much of the power of `reduce`, and yet the resulting language retains enough power to express intractable problems. Indeed, we now show how to express the Hamiltonian path problem, using boolean-only `reduce`.

The Hamiltonian path problem is defined as follows: Given a graph G, is there a path p in G such that p visits each node of G exactly once? This problem is also known to be NP-complete [20]. This problem is solved by the following query.

```
MATCH (n)
WITH collect(n.name) AS allNodes
MATCH path=(:Start)-[*]-()
WITH path, allNodes,
    [y IN nodes(path) | y.name] AS nodesInPath
WHERE all(node in allNodes where node IN nodesInPath)
AND size(allNodes)=size(nodesInPath)
RETURN path LIMIT 1
```

The run of this query is illustrated by Fig. 3. The first two lines of the query collect all node ids into the list `allNodes` that contains `n1,n2,n3,n4,n5` in some order. Then the query matches any trail path from `Start` to `End` nodes (`n1` and `n5`) and collects its node ids into the list `nodesInPath`. One such path is shown by thick black arrows in the figure, with the resulting list `nodesInPath`

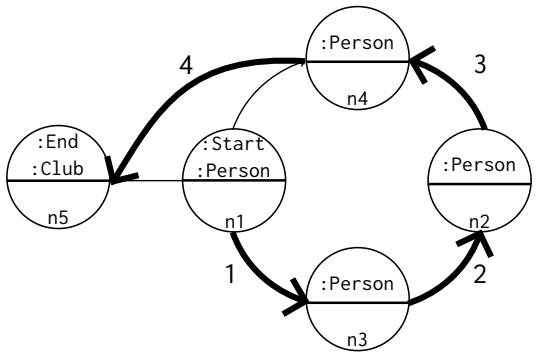


Fig. 3. A Hamiltonian path in a simplified cycling graph

being `[n1,n2,n3,n4,n5]`. To check that each node is traversed at least once we use the `all` (a specialization of `reduce` that checks whether a predicate holds for all elements of a list), and apply it on `allNodes` to check that each of its elements appears in `nodesInPath`. To check that each node is traversed at most once we ensure that the sizes of `allNodes` with `nodesInPath` are the same, which is the case here.

Thus, even with severe restrictions on `reduce`, one can encode intractable problems in Cypher. There is a worrying element here from the language design point of view: it is *how easy* it was to write these queries.

This now leads us to two questions. First, will this theoretical complexity show up in practice? After all, there are many NP-hard problems that are routinely solved even in the database context, not least the problem of finding trail paths. This is an NP-hard problem and it has not stopped Cypher from being the (so far) dominant graph query language, despite trails being its default path semantics. There are multiple other examples of this kind, like exponential-time-hard typechecking problems in widely used programming languages [29] or the success of satisfiability checkers [30]; in all of these the worst case behavior rarely manifests itself in everyday programming practice.

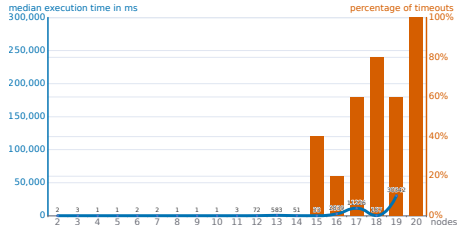
The second question is how likely the badly behaving queries are to appear in applications. Will the average of moderately advanced programmer write such queries, and will they be able to predict their behavior.

The next two sections answer these questions. First, we do an experimental study to show that theoretically intractable queries from this section are completely impossible, and timeout even on tiny graphs. Second, we do a user study to discover that the key elements that made queries intractable are not only routinely taught to programmers as viable techniques, but that programmers in addition do not correctly estimate their cost (erring by several orders of magnitude).

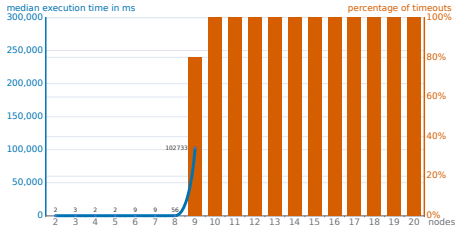
6 Lists are terrible

While NP-hardness per se is often bad news, there are exceptions when the cases witnessing hardness do not realistically occur. Thus, to see whether the results of the previous section indeed spell bad news for list processing in graph queries, we evaluate such queries experimentally, with the result fully expected by the reader who has glanced at the title of the section.

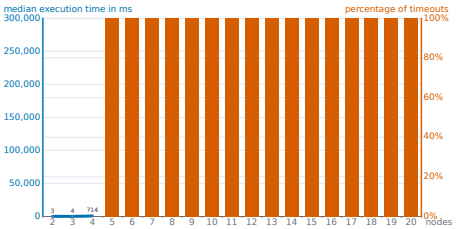
To test the actual performance of hard queries, we ran a set of tests using Neo4j, the most widely used graph database engine. We started our experiments with randomized data, and discovered that the performance was so poor (only very small graphs with fewer than 30 nodes could be handled before timeout) that there was simply no real data of such tiny size we could realistically extend



(a) Median execution time and number of time-outs for $p = 0.1$



(b) Median execution time and number of time-outs for $p = 0.3$



(c) Median execution time and number of time-outs for $p = 0.8$

Fig. 4. Results of the performance tests on Neo4j for the Hamiltonian path problem

the experiments to. 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.

The tests proceed as follows. For each $p \in \{0.1, 0.2, \ldots, 1.0\}$ and each $n \in [\texttt{minNodes}, \texttt{maxNodes}]$, do the following five times[3]:

(1) Generate a random graph with n nodes, with the value of property `name` ranging from 1 to n, in which each pair of nodes is connected by an edge with probability p.
(2) Assign the labels `Start` and `End` to one random node each.
(3) Do the following five times and log the execution time of the last four iterations (the first iteration gives Neo4j a chance to generate the appropriate indices):
(4) Generate an instantiation of the chosen query
(5) Run the query on the generated graph, and declare it timed-out after five minutes.

Figure 4 presents a sample of runtimes of the Cypher Hamiltonian path query. We show it here for small edge probabilities $p = 0.1,\ 0.3$ and one high probability 0.8 (the url with supplementary material contains full results). As p increases, the graph becomes closer to a complete graph. The number of paths then grows as the factorial function, quickly degrading performance.

The figure shows the median execution time (blue line, scale on the left) and percentage of timeouts (bars; scale on the right). If some runs time out, we take the median only over those that do not (explaining nonmonotone behavior in cases where we see both timeouts and successful executions).

According to these tests, for 65.26% of the configurations all iterations time out. When edge probability is over 0.7, graphs with more than 4 nodes cannot be handled within 5min running time. It is not much better for sparser graphs, with timeouts on 10 nodes for $p = 0.3$ and 15 nodes

[3]To avoid memory pollution between rounds, the Neo4j server was manually restarted between each increase of p.

for $p = 0.1$. The biggest graph for which the Hamiltonian path problem was solved contains just 19 nodes. Further, if a solution was found within the prescribed time, it was rather slow (e.g., over 3min for $p = 0.6$ and only 6 nodes).

Of course there could be another potential culprit, namely trail semantics, which is the default semantics of Cypher: only paths with no repeated edges are returned. In fact this makes Cypher pattern matching NP-complete in general [19] though it is not the case for all queries. Indeed, [31] identified the class $\mathrm{T}_{\mathrm{tract}}$ of tractable queries for the regular trail query problem. Queries from $\mathrm{T}_{\mathrm{tract}}$ are in NL (and thus PTIME) and queries not in $\mathrm{T}_{\mathrm{tract}}$ are NP-complete. In fact, relatively few real-life regular patterns fall outside $\mathrm{T}_{\mathrm{tract}}$ which explains the good behavior of the trail semantics in practice [10]. A rare pattern that does occur in practice and has a theoretical NP-complete bound is A^*BA^*, i.e., a path of edges labeled by A with the exception of a single edge labeled B that occurs anywhere on the path. The existence of such a path between given start and end nodes is of course easily checked in Cypher:

```
MATCH p = (:Start)-[:A*]->()-[:B]->(:End)-[:A*]->()
RETURN p
LIMIT 1
```

Thus, as the first test to see whether the culprit of the bad behavior of the Hamiltonian path query is lists or trails, we test the performance of this query looking for A^*BA^* trails.

For the sparsest graphs with $p = 0.1$, where for Hamiltonian paths we witnessed the 100% timeout rate at 20 nodes, here we observed a very good performance with queries taking $\leq$1ms with out of memory errors appearing on graphs three times larger than those witnessing timeouts for Hamiltonian path. With the increased density ($p = 0.3$), where Hamiltonian path could not be handled on graphs with 10 or more nodes, we see again that the A^*BA^* query performs well on graphs up to three times the size, with similar $\leq$1ms running times and out of memory errors from 32 nodes. Finally, for dense graphs ($p = 0.8$), we have a similar picture: out of memory errors on 24 nodes (6 times larger than the largest graph handled for the Hamiltonian path query), with 50% of timeouts on 21 nodes. Note that for all the runs that did not result in a timeout or out of memory errors, the execution time was $\leq$1ms. This very large gap between the performance of two theoretically NP-complete queries, one using lists and trails and the other using trails alone, points to lists as the key reason for poor performance.

To further confirm that lists, rather than trails, are the real cause of extremely poor behavior, we test the subset sum query from Section 5. Recall that this query also encodes an NP-complete problem but does so with *shortest paths* rather than trails, and of course finding shortest paths is tractable.

The results of the performance tests of the subset sum query on Neo4j are shown in Figure 5, (a). Recall that in our encoding of this problem as a line graph with parallel edges, the graph is fixed (there is no random generation), so we only report the number of nodes on the x-axis. The y-axis shows the median running time with the same 5min timeout, and the right bar indicates the percentage of timeouts. This time we do 20 iterations for each length. Although the performance is good for very small graphs, staying under 2000ms for up to 20 nodes, the exponential nature of the problem becomes very quickly apparent, eventually reaching the 100% timeout rate on 27 nodes.

Since for this query the only source of complexity is the use of lists to encode the NP-hard problem subset sum, together with other results of this section it clearly points to complete inability of the state-of-the-art graph database engine to handle anything other than the tiniest of inputs.

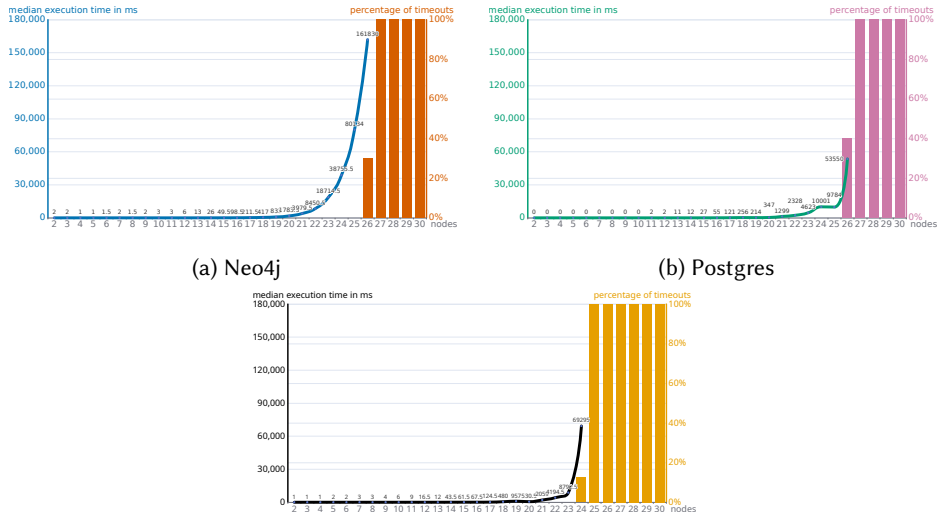


Fig. 5. Results of the performance tests for the subset sum problem

7 How realistic are these queries? A user study

While the queries studied here cannot be handled by graph database engines on realistic size graphs, one can legitimately ask how likely are they to be written by programmers. They are rather short and simple in appearance, but they combine several key elements – pattern matching, the use of lists, and `reduce` in filtering conditions – that perhaps would not all be known to or used with ease by a moderately advanced programmer. Or perhaps when programmers write such queries, they will quickly recognize how problematic they are?

Our goal now is to address these questions. We argue that (1) programmers are taught to write such queries, (2) they write them with relative ease; and (3) they do not anticipate the complexity of such queries.

For the first point, we have ample evidence that this style of writing queries (using lists, `reduce`, and conditions based on them) is advocated by various sources programmers refer to, such as books and multiple blogpost directed to programmers rather than academics. We list here a few examples. The main textbook reference on Neo4j and Cypher programming [37] provides examples based on typical customer problems; one of them is finding a shortest delivery route (page 139). As the way to handle such queries, [37] suggests using lists and `reduce` both in filters and outputs. Likewise, [40] advocates using `reduce` and lists (specifically with Boolean conditions as we do) in `WHERE`. Amazon Neptune documentation [6] explains how to mimic `ALL`, `NONE`, and `ANY`, all essential in our list queries and sufficient to produce intractable ones. At this point Neptune supports core Cypher; these features were deemed important to Amazon to include them in their documentation for programmers. Another example is [9] which is devoted to teaching programmers how to use `reduce`; a similar point is made in [34] which explains the ubiquitous `WITH` clause to Cypher programmers, using `reduce` over lists as an example. Queries in these references are similar in structure to the ones we use; in fact some of them are quite a bit more complex than ours.

While for point (1) it is easy to provide resources that directly support it, for points (2) and (3) this could only realistically be done with a user study, to demonstrate that mainstream developers

would be at ease with such features. To test our hypothesis, we surveyed 45 MSc students who had followed a course on graph databases and Cypher at Université Paris Cité, and successfully passed an exam. We gave them six questions, with the first two, asking for typical Cypher `MATCH` queries, used to evaluate their knowledge of the subject, and the remaining four delving into lists and `reduce`. The average grade for the first two questions was 6/10, with many students doing very well, thus showing their suitability for subsequent questions.

The third question asked them to explain the Hamiltonian path query we studied here: specifically we asked them what it returns, and how it works. Correct answer was given by 78%, with incorrect ones occurring mainly among those respondents who did not do well on basic `MATCH` queries already.

The next question was quite revealing: we asked them to estimate on how many nodes we will start seeing timeouts. Recall that our experiements show the cutoff for timeout is well below 100 nodes. As options we gave them intervals between 10^n and 10^{n+1} for n between 1 and 5 (with the correct answer being $n = 1$). The average answer in our study is 3.49, i.e., the respondents overestimate the responsiveness of graph DBMSs on such queries by several orders of magnitude. Specifically, only 7% gave correct answer, while $n = 2$ was chosen by 12%, $n = 3$ by 27%, $n = 4$ by 32%, and $n = 5$ by 22%. Interestingly enough, the three respondents who gave the correct answer were the very best students in class, i.e., truly expert users, as opposed to others, who were by and large very competent programmers.

In question 5 we asked how natural the Hamiltonian path query was looking to them and if they would feel at ease with writing similar ones in a corporate environment. The vast majority of students found the query perfectly clear (some pointing out that being very similar to the map and fold operations of programming languages made this query easy to write and understand). Some suggested that the query may require too many paths to explore, and therefore it would be safer to rely on well tested library functions for running it on larger graphs, although the previous question indicates they did not correctly estimate where the dangerous zone is. Finally, in question 6 we asked them to write the subset sum query. A perfectly correct answer was given by 55% of respondents, while 18% gave answers that would be correct after some minor debugging, and only 27% could not solve the problem, again correlating with how well they have done on basic Cypher queries.

The results of our survey suggest that while true expert programmers (a small minority) would realize that there might be performance issues in queries using lists, the majority would not be aware of the problem, while being perfectly capable of understanding and writing problematic queries.

8 Can SQL help?

A case for the use of relational database engines over specialized graph engines has been made in the context of analytic and concurrent transactional workloads [16, 33, 35]. On the other hand, the idea that graph databases outperform relational databases for navigational queries is widespread, even though reservations have been expressed in the case of complex queries [3, 25]. Following pointers, as described in [43], can indeed be done in constant time, thereby avoiding costly joins, which suggests an advantage for native graph structures.

Since no crystal clear picture emerges from existing studies as to the advantages of a relational representation or a native graph engine, we look at the problematic queries from the previous sections and see how SQL DBMSs would handle them. We ran the same set of tests as for Neo4j on Postgres and DuckDB, to analyze performance on DBMSs oriented towards OLTP and OLAP workloads. All the queries from the previous sections can be expressed in SQL with recursive common table expressions.

There are multiple ways to encode property graphs in relational structures. We settle for a simple encoding to minimize the number of joins and facilitate writing queries.

For the A^*BA^* query, which is intended to show how well the trail semantics is enforced, we encode edges as ternary relations with string type attributes for source, target, and label. To store paths, we use the built-in array type: a path with edges $(s_0, t_0), \ldots, (s_k, t_k)$, where s_is and t_is are sources and targets of edges, is encoded as an array $[s_0.t_0, \ldots, s_k.t_k]$, where $s_i.t_i$ is the concatenation of two strings. The recursive part of the query then constructs A-labeled paths, by augmenting their length and arrays representing the paths, while ensuring the trail condition in the `WHERE` clause by checking that newly added edges do not appear among those already on the path. Since the number of trails is finite, this recursion terminates. Finally, we concatenate an A-labeled trail, a B-edge, and another A-labeled trail while checking that the two trails do not overlap. The encoding also uses relations `SNode` and `ENode` with a single attribute `node` for instantiating start and end nodes of paths.

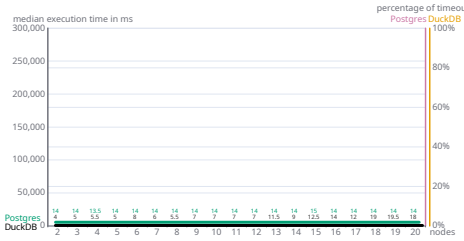
The query testing for the existence of such a path is shown below:

```
WITH RECURSIVE a_kleene_star AS (
    SELECT s, t, 0 AS depth, array[s,t] AS path,
           array[s||'.'|| t] AS edges FROM A
    UNION
    SELECT A.s, A.t, a_kleene_star.depth+1,
           a_kleene_star.path||A.t,
           a_kleene_star.edges ||
           concat(A.s||'.',A.t)
    FROM A, a_kleene_star
    WHERE A.s=a_kleene_star.t AND
    NOT concat(A.s||'.',A.t)=any(a_kleene_star.edges)
    )
SELECT A1.s, A2.t
FROM a_kleene_star A1, a_kleene_star A2, B
WHERE A1.s=SNode.node AND A2.t=ENode.node
      AND A1.t=B.s AND B.t=A2.s
      AND NOT (A1.edges && A2.edges)
LIMIT 1
```

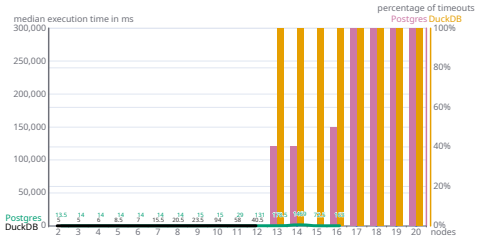
To test the performance, we use the same data as for the Cypher query, modulo the representation of data as relations. The performance of this query on Postgres and DuckDB is shown in Figure 6, as a green line and pink bars for Postgres and a black line and yellow bars for DuckDB.

Unlike Neo4j, both Postgres and DuckDB exhibit a similar behavior, and both struggle with this query. In fact, for $p = 0.3$, we see the 100% timeout rate on graphs of 17 nodes; Neo4j could handle graphs twice the size. Of course the A^*BA^* query is a typical graph pattern matching query on which graph DBMS are expected to optimize better than relational ones.

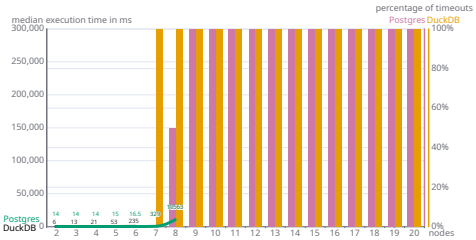
We next move to the Hamiltonian path query. Since it only concerns the underlying graph structure (no reference to properties), we use an even simpler encoding of the graph as a binary relation `G` with attributes `src` and `tgt` ranging over node identifiers of sources and targets of edges. Candidate Hamiltonian paths are constructed iteratively by initially storing the source and target of each edge in a different array. Whenever another edge can be reached from an edge previously stored in the array, the target of that new edge is added to the array if it was not already in it. At the end of the iteration we check whether one of those arrays is of the same size as the graph,



(a) Median execution time and number of timeouts for $p = 0.1$



(b) Median execution time and number of timeouts for $p = 0.3$



(c) Median execution time and number of timeouts for $p = 0.8$

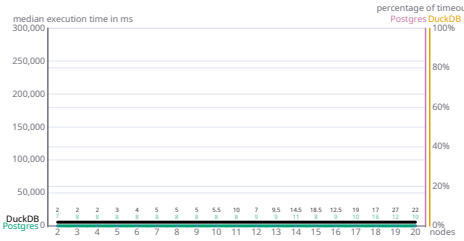
Fig. 6. Results of the performance tests on Postgres and DuckDB for A^*BA^*

indicating the presence of a Hamiltonian path. The query tests for the existence of a Hamiltonian path, and hence we can stop when one was found.

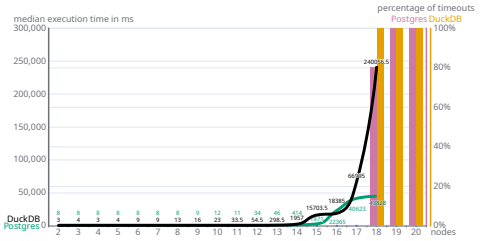
```
WITH RECURSIVE paths(startP, endP, path)
AS (SELECT src AS startP, tgt AS endP,
           ARRAY[src,tgt] AS path
    FROM G
    UNION
    SELECT startP, tgt, array_append(path,tgt)
    FROM G, paths
    WHERE src=endP AND tgt <> ALL(path))
SELECT * FROM paths
WHERE ARRAY_LENGTH(path,1) =
      (SELECT COUNT(DISTINCT src) FROM G)
LIMIT 1
```

Performance results for this query are shown in Figure 7. While for low probabilities and small values of n Neo4j and both relational systems perform well (with relational being marginally slower), SQL's coverage is better when it comes to timeouts on a higher number of nodes (e.g., 100% timeout on 19 nodes for $p = 0.3$ as opposed to 10 for Cypher). This is most likely again due to list processing in the Cypher query, that precludes optimizations and forces the engine to find all paths; giving advantage to relational optimizations. Regardless, just like Cypher, SQL can only handle tiny graphs here, and for any reasonable size graph performance would not be adequate independently of the choice of SQL or Cypher to encode the problem.

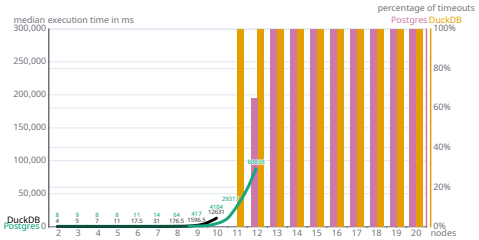
We finally look at the subset sum query. Here we encode the graph as a ternary relation `G` storing the source, target and weight of each edge. Candidate paths are iteratively constructed as array



(a) Median execution time and number of timeouts for $p = 0.1$



(b) Median execution time and number of timeouts for $p = 0.3$



(c) Median execution time and number of timeouts for $p = 0.8$

Fig. 7. Results of the performance tests on postgres and duckDB for the Hamiltonian path problem

structures. For each edge in the graph, an array containing its source identifier, weight and target identifier is first initialized. Variables for the first node in the path, the last node, and the total weight in the path are also initialized. When another edge can be reached from the edge previously stored in the array, the target of that new edge is added to the array at each iterative stage. As we only run this query on graphs with no cycles (see Figure **??**), the computation terminates and only gives rise to trails. Finally, we only keep path with the total weight 0, ending on a prescribed target node $T, to test for different graph sizes:

```
WITH RECURSIVE paths(p_src, p_tgt, path, total_weight)
AS (SELECT src as p_src, tgt as p_tgt,
           ARRAY[src,weight,tgt] as path,
           weight as total_weight
    FROM G
    WHERE src = 0
    UNION
    SELECT p_src, tgt,
        array_append(array_append(path,weight),tgt),
        total_weight+weight as total_weight
    FROM G, paths
    WHERE src=p_tgt)
SELECT * FROM paths
WHERE total_weight=0 and p_src=0 and p_tgt=$T
```

Performance results for this query are shown in Fig. 5. Here Cypher and both SQL implementations we tested behave very similarly, with the exponential growth starting a bit earlier for Cypher, but then 100% of timeouts is reached on graphs of the same size.

To sum up, a SQL DBMS, whether tuned for OLAP or OLTP workloads, does not outperform a native graph DBMS on the problematic queries we explored: their performance is roughly comparable.

There is one striking difference however. Queries such as subset sum and Hamiltonian path that used lists explicitly are *very easy* to write in Cypher, and are likely to be written in Cypher, as we demonstrated in Section 7. In SQL on the other hand these queries are much harder to write, they require a combination of recursion and arrays, and their general shape may serve as an indication that their performance might not be adequate. Therefore it is ease of use of list operations in Cypher queries as its specific design feature that can lead to significantly degraded performance.

9 Lessons for language design

While performance figures in Section 6 may suggest that list processing can ruin everything, it is nonetheless a very convenient device that oftentimes can and will be used without causing significant problems. The main culprit behind the poor performance is not a particular database engine but rather the *design of the language* that makes it possible to write offending queries with ease and without being aware of their performance issues.

Before outlining a possible remedy, note that a careless approach to the design of language features and their semantics can lead to even worse circumstances. A prominent example of this is the initial design of SPARQL pattern matching that was changed after its complexity consequences were discovered [5, 28]. GQL is not immune from this. Consider the following example inspired by [17]. The database is a very simple graph: it has a loop on a node with label `lab1`, and an isolated node with label `lab2` and three properties `a,b,c` with integer values. Next consider the query

```
MATCH p=allShortestPaths((:lab1)-[*]->()), (y:lab2)
WITH reduce(s=0,
            v IN [r IN relationships(p) | r.v] | s+v) AS x,
     y.a AS a, y.b AS b, y.c AS c
WHERE a*x*x + b*x + c=0
RETURN p LIMIT 1
```

There are two ways of providing a semantics to this query, depending on the point at which the filter in `WHERE` is applied:

- if it is a post-filter, i.e., applied after *shortest*, then a single shortest path of length 1 is found, and the condition simply checks if $a + b + c = 0$;
- if it is a *pre-filter*, i.e., *shortest* applies to paths that satisfy the condition, then it checks whether the quadratic equation $ax^2 + bx + c = 0$ has a positive integer solution.

The latter means that using `reduce` in pre-filters one can check conditions that are at best done by specialized solvers but in general might even be undecidable. Indeed, if instead of checking the existence of an integer solution to a univariate quadratic polynomial we asked for an integer solution to a multivariate polynomial of degree 4 with a fixed number of variables, this would be an undecidable problem [21], yet encodable with pre-filters.

If we look at all our examples that led to high complexity of queries, they used post-filters with conditions involving outputs of `reduce`. The above example shows that a simple looking language design decision – using pre-filters instead of post-filters – can make the problem much worse and even lead to undecidability of query answering. The question therefore is:

what are the lessons for query language design for graph databases?

Based on the findings of this paper, we identify three main lessons.

The **first lesson** is that conditions based on `reduce` should be *disallowed* in `WHERE`. In fact, it is easy to trace subexpressions used in `WHERE`, and if any of them used `reduce` in its syntax tree, such

an expression should result in a compilation error. This applies even to the strongest restriction of Section 5 when `all` and `size` are the only allowed instances of `reduce`.

As indicated in Section 4, there are multiple examples showing the usefulness of `reduce`. In particular, doing a computation on lists and then returning results rather than using them for filtering is both useful and harmless complexity-wise as long as lists of lists are not produced. Thus, the **second lesson** is that using `reduce` is fine in `RETURN`, `WITH` and similar statements in other languages, as long as there is no violation of lesson one, and output of `reduce` is either a scalar value (restriction 1 of Section 5) or even more liberally a list of scalar values.

This leaves us with an interesting case of using `reduce` to simulate the power of various automata. Our **lesson three** is that for such problems, query languages should provide facilities that *are not based on* `reduce`. This has already been done for RPQs, with GQL and SQL/PGQ providing facilities for expressing them. For more complex path queries, such as RPQs with data or extended CRPQs, so far GQL provides ad hoc facilities. However, it is already recognized that the language falls short of some desired expressiveness, and work is under way to enhance GQL's capabilities ahead of the next release. Our results inform this effort by ruling out an otherwise tempting solution of copying existing facilities of Cypher despite their significant negative consequences.

Acknowledgments

This work was supported by ANR Project VeriGraph ANR-21-CE48-0015 (Leonid Libkin), a grant from RelationalAI to the IRIF laboratory (Leonid Libkin) and Poland's National Science Centre grant 2018/30/E/ST6/00042 (Alexandra Rogova).

References

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

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

[3] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep Lluís Larriba-Pey. 2013. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013, co-located with SIGMOD/PODS 2013, New York, NY, USA, June 24, 2013*, Peter A. Boncz and Thomas Neumann (Eds.). CWI/ACM, 15.

[4] Kemafor Anyanwu and Amit Sheth. 2003. P-Queries: enabling querying for semantic associations on the semantic web. In *Proceedings of the 12th International Conference on World Wide Web* (Budapest, Hungary) *(WWW '03)*. Association for Computing Machinery, New York, NY, USA, 690–699. https://doi.org/10.1145/775152.775249

[5] 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 *Proceedings of the 21st World Wide Web Conference 2012, WWW 2012, Lyon, France, April 16-20, 2012*, Alain Mille, Fabien Gandon, Jacques Misselis, Michael Rabinovich, and Steffen Staab (Eds.). ACM, 629–638. https://doi.org/10.1145/2187836.2187922

[6] AWS. 2024. Rewriting Cypher queries to run in openCypher on Neptune. https://docs.aws.amazon.com/neptune/latest/userguide/migration-opencypher-rewrites.html.

[7] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Trans. Database Syst.* 37, 4, Article 31 (dec 2012), 46 pages. https://doi.org/10.1145/2389241.2389250

[8] Chris Barrett, Riko Jacob, and Madhav Marathe. 2000. Formal-Language-Constrained Path Problems. *SIAM J. Comput.* 30, 3 (May 2000), 809–837. https://doi.org/10.1137/S0097539798337716

[9] Vlad Batushkov. 2019. Neo4j Reduce in Action. https://vladbatushkov.medium.com/one-month-graph-challenge-radio-3591d01674b8.

[10] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *Proc. VLDB Endow.* 11, 2 (2017), 149–161. https://doi.org/10.14778/3149193.3149196

[11] Mariano P. Consens and Alberto O. Mendelzon. 1990. GraphLog: A Visual Formalism for Real Life Recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Nashville,

Tennessee, USA) *(PODS '90)*. Association for Computing Machinery, New York, NY, USA, 404–416. https://doi.org/10.1145/298514.298591

[12] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A Graphical Query Language Supporting Recursion. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 323–330. https://doi.org/10.1145/38713.38749

[13] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2246–2258.

[14] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020*. ACM, 377–392. https://doi.org/10.1145/3318464.3386144

[15] Michael Elberfeld, Andreas Jakoby, and Till Tantau. 2010. Logspace Versions of the Theorems of Bodlaender and Courcelle. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. 143–152. https://doi.org/10.1109/FOCS.2010.21

[16] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M. Patel. 2015. The Case Against Specialized Graph Analytics Engines. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org.

[17] 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 *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Floris Geerts, Hung Q. Ngo, and Stavros Sintos (Eds.). ACM, 241–250. https://doi.org/10.1145/3584372.3588662

[18] 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. In *26th International Conference on Database Theory, ICDT 2023, March 28-31, 2023, Ioannina, Greece (LIPIcs, Vol. 255)*, Floris Geerts and Brecht Vandevoort (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:22. https://doi.org/10.4230/LIPIcs.ICDT.2023.1

[19] 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 *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1433–1445. https://doi.org/10.1145/3183713.3190657

[20] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA.

[21] William Gasarch. 2021. Hilbert's Tenth Problem for Fixed d and n. *Bull. EATCS* 133 (2021). http://eatcs.org/beatcs/index.php/beatcs/article/view/643

[22] Amélie Gheerbrant, Leonid Libkin, Liat Peterfreund, and Alexandra Rogova. 2025. GQL and SQL/PGQ: Theoretical Models and Expressive Power. *Proc. VLDB Endow.* 18, 6 (2025). https://arxiv.org/abs/2409.01102

[23] Gösta Grahne and Alex Thomo. 2004. Query Answering and Containment for Regular Path Queries under Distortions. In *Foundations of Information and Knowledge Systems, Third International Symposium, FoIKS 2004, Wilhelminenberg Castle, Austria, February 17-20, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 2942)*, Dietmar Seipel and Jose Maria Turull Torres (Eds.). Springer, 98–115. https://doi.org/10.1007/978-3-540-24627-5_8

[24] Stéphane Grumbach and Tova Milo. 1996. Towards Tractable Algebras for Bags. *J. Comput. Syst. Sci.* 52, 3 (1996), 570–588. https://doi.org/10.1006/JCSS.1996.0042

[25] Petri Kotiranta, Marko Junkkari, and Jyrki Nummenmaa. 2022. Performance of Graph and Relational Databases in Complex Queries. *Applied Sciences* 12, 13 (2022).

[26] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. 2016. Querying Graphs with Data. *J. ACM* 63, 2, Article 14 (mar 2016), 53 pages. https://doi.org/10.1145/2850413

[27] Leonid Libkin and Limsoon Wong. 1997. Query Languages for Bags and Aggregate Functions. *J. Comput. Syst. Sci.* 55, 2 (1997), 241–272. https://doi.org/10.1006/JCSS.1997.1523

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

[29] Harry G. Mairson. 1989. Deciding ML typability is complete for deterministic exponential time. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 382–401. https://doi.org/10.1145/96709.96748

[30] Sharad Malik and Lintao Zhang. 2009. Boolean satisfiability from theoretical hardness to practical success. *Commun. ACM* 52, 8 (2009), 76–82. https://doi.org/10.1145/1536616.1536637
[31] Wim Martens, Matthias Niewerth, and Tina Popp. 2023. A Trichotomy for Regular Trail Queries. *Log. Methods Comput. Sci.* 19, 4 (2023). https://doi.org/10.46298/LMCS-19(4:20)2023
[32] Alberto O. Mendelzon and Peter T. Wood. 1989. Finding Regular Simple Paths in Graph Databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases (VLDB)*, Peter M. G. Apers and Gio Wiederhold (Eds.). Morgan Kaufmann, 185–193. http://www.vldb.org/conf/1989/P185.PDF
[33] Marc Najork, Dennis Fetterly, Alan Halverson, Krishnaram Kenthapadi, and Sreenivas Gollapudi. 2012. Of hammers and nails: an empirical comparison of three paradigms for processing large graphs. In *Proceedings of the Fifth International Conference on Web Search and Web Data Mining, WSDM 2012, Seattle, WA, USA, February 8-12, 2012*, Eytan Adar, Jaime Teevan, Eugene Agichtein, and Yoelle Maarek (Eds.). ACM, 103–112.
[34] Mark Needham. 2013. Neo4j/Cypher: Getting the hang of the WITH statement. https://www.markhneedham.com/blog/2013/03/20/neo4jcypher-getting-the-hang-of-the-with-statement/.
[35] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases?: Benchmarking Real-Time Social Networking Applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*, Peter A. Boncz and Josep Lluís Larriba-Pey (Eds.). ACM, 12:1–12:7.
[36] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. 2017. Regular Queries on Graph Databases. *Theor. Comp. Sys.* 61, 1 (July 2017), 31–83. https://doi.org/10.1007/s00224-016-9676-2
[37] Ian Robinson, James Webber, and Emil Efrem. 2013. . O'Reilly Media.
[38] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2017), 420–431. https://doi.org/10.1145/3186728.3164139
[39] Jacques Sakarovitch. 2009. *Elements of Automata Theory.* Cambridge University Press.
[40] Cristian Scutaru. 2024. 10 Brilliant or Atrocious Neo4j Cypher Hacks. https://data-xtractor.com/blog/databases/neo4j-cypher-hacks/.
[41] Petra Selmer. 2019. Existing Languages Working Group: GQL influence graph. https://www.gqlstandards.org/existing-languages.
[42] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *GRADES*. ACM, 7.
[43] Jim Webber, Emil Eifrem, and Ian Robinson. 2013. *Graph databases.* O'Reilly Media, Incorporated.

Received October 2024; revised January 2025; accepted February 2025