PathFinder: Returning Paths in Graph Queries

Benjamín Farías[1,2], Wim Martens[3], Carlos Rojas[2], and Domagoj Vrgoč[1,2(✉)]

[1] Pontificia Universidad Católica de Chile, Santiago, Chile
{bffarias,vrdomagoj}@uc.cl
[2] IMFD Chile, Santiago, Chile
cirojas6@uc.cl
[3] University of Bayreuth, Bayreuth, Germany
wim.martens@uni-bayreuth.de

Abstract. Path queries are a central feature of all modern graph query languages and standards, such as SPARQL, Cypher, SQL/PGQ, and GQL. While SPARQL returns *endpoints* of path queries, it is possible in Cypher, SQL/PGQ, and GQL to return *entire paths*. In this paper, we present the first framework for returning paths that match regular path queries under all fifteen modes in the SQL/PGQ and GQL standards. At the core of our approach is the product graph construction combined with a way to compactly represent a potentially exponential number of results that can match a path query. Throughout the paper we describe how this approach operates on a conceptual level and provide runtime guarantees for evaluating path queries. We also develop a reference implementation on top of an existing open-source graph processing engine, and perform a detailed analysis of path querying over Wikidata to gauge the usefulness of our methods in a real world scenario. Compared to several modern graph engines, we obtain order-of-magnitude speedups and remarkably stable performance, even for theoretically intractable queries.

1 Introduction

Graph databases [4,40] have gained significant popularity and are used in areas such as Knowledge Graph management [25], Semantic Web applications [5] and Biology [29]. The query languages for handling graph data include SPARQL [24], a W3C standard for querying RDF, and Cypher, GQL and SQL/PGQ [14,21], which are part of the ISO initiative to standardize querying over property graphs.

All these languages have *path queries* as a core feature. In SPARQL, these are supported through *property paths*, which are a variant of *regular path queries* (*RPQs*) [6,11,13,36]. Intuitively, an RPQ is an expression $(?x, \texttt{regex}, ?y)$, where `regex` is a regular expression and $?x, ?y$ are variables. When evaluated over an edge-labeled graph G (e.g., RDF or property graph), it extracts all node pairs (`n1`, `n2`) such that there is a path in G from `n1` to `n2`, whose edge labels form a word that matches `regex`. However, SPARQL property paths only return *endpoints* of paths. In important applications such as money laundering detection or in investigative journalism [14], retrieving the path is of crucial importance.

G. Demartini et al. (Eds.): ISWC 2024, LNCS 15232, pp. 135–154, 2025.
https://doi.org/10.1007/978-3-031-77850-6_8

To accommodate this use case, Cypher, GQL, and SQL/PGQ all extend RPQs with a mechanism for returning paths. To avoid having to return an infinite number of paths (which can happen if the graph is cyclic), these languages let the user specify restrictions on the type of paths to be returned. These are combinations of e.g., *simple* (no duplicate nodes), *trail* (no duplicate edges), *shortest* (shortest paths that match the expression), or *any* (just return a single, arbitrarily chosen path).

While Cypher, GQL, and SQL/PGQ all recognize the need to return paths that match regular path queries, the support for these features in modern graph databases is surprisingly lacking. As already stated, in SPARQL engines, RPQ matching is fully supported, but paths cannot be returned. In property graph engines that implement (fragments of) GQL or SQL/PGQ, only a handful of path modes is supported, and no property graph engine to date supports all regular path queries like SPARQL engines do. For reference, we summarize the support for path features in several popular SPARQL and property graph engines in the table below. Here, "RPQ" means the support for all regular path queries, while the other columns signal which types of paths can be returned.

	RPQ	WALK	TRAIL	SIMPLE	ACYCLIC	SHORTEST
BlazeGraph [44]	✓	–	–	–	–	–
Jena [42]	✓	–	–	–	–	–
Virtuoso [15]	✓	–	–	–	–	–
Neo4j [49]	partial	✓	✓	–	–	✓
Nebula [45]	partial	partial	✓	–	✓	✓
Memgraph [43]	partial	✓	✓	–	–	✓
Kuzu [28]	partial	✓	–	–	–	✓
DuckPGQ [51]	partial	✓	–	–	–	✓
PathFinder	✓	✓	✓	✓	✓	✓

We see that SPARQL engines support all RPQs, but cannot return paths. On the other hand, while several property graph engines can return some (but not all) types of paths, they only offer a limited support for regular expressions. In addition, both SPARQL engines and property graph engines have significant issues handling path queries [38,48]. We believe that this is the case because of scalability, that is, *the number of paths that match an RPQ can be exponential in the size of the graph* [31].

Our Contribution. To handle the aforementioned issues, in this paper we introduce PathFinder, a unified framework for returning paths in graph query answers. PathFinder supports returning paths, using *all 15 path modes* prescribed in the GQL and SQL/PGQ standards, while at the same time permitting *all RPQs*. To achieve this, we show how classical graph search algorithms can be adapted to run on the product graph [31,36] and at the same time produce a compact representation of all paths that need to be returned. This construction is done lazily and the paths can be returned as soon as they are detected.

We test our approach over Wikidata [47] using real world queries [30]. Our experiments show two interesting things; namely, that over the real-world data:

- returning up to 100.000 paths witnessing RPQ answers can be done in the same time that other engines need to determine the existence of a path; and
- automata-based approaches for RPQ evaluation can scale well and even significantly outperform regular expression based algorithms for RPQ evaluation while also returning paths.

2 Graph Databases and Path Queries

Graph Databases. Let Nodes be a set of node identifiers and Edges be a set of edge identifiers, with Nodes and Edges being disjoint. Additionally, let Lab be a set of labels. Following [5,14,20,31], we define graph databases as follows.

Definition 1. *A* graph database G *is a tuple* (V, E, ρ, λ)*, where*

- $V \subseteq$ *Nodes is a finite set of nodes,*
- $E \subseteq$ *Edges is a finite set of edges,*
- $\rho : E \to (V \times V)$ *is a total function, and*
- $\lambda : E \to$ Lab *is a total function assigning a label to each edge.*

Intuitively, $\rho(e) = (v_1, v_2)$ means that e is a directed edge going from v_1 to v_2. Here we use a simplified version of property graphs which omits node and edge properties (with their associated values). This is done since the type of queries we consider only use nodes, edges, and edge labels. However, all of our results transfer verbatim to the full version of property graphs.

Paths. A sequence $p = v_0 e_1 v_1 e_2 v_2 \cdots e_n v_n$ is called a *path* in a graph database $G = (V, E, \rho, \lambda)$, if $n \geq 0$, $e_i \in E$, and $\rho(e_i) = (v_{i-1}, v_i)$ for $i = 1, \ldots, n$. If p is a path in G, we write $\mathsf{lab}(p)$ for the sequence of labels $\mathsf{lab}(p) = \lambda(e_1) \cdots \lambda(e_n)$ occurring on the edges of p. We write $\mathsf{src}(p)$ for the starting node v_0 of p, and $\mathsf{tgt}(p)$ for the end node v_n of p. The length of a path p, denoted $\mathsf{len}(p)$, is defined as the number n of edges it uses. We say that a path p is a

- WALK, for every p (as per GQL and SQL/PGQ terminology);
- TRAIL, if p does not repeat an edge ($e_i \neq e_j$ for every $i \neq j$);
- ACYCLIC, if p does not repeat a node ($v_i \neq v_j$ for every $i \neq j$); and
- SIMPLE, if p does not repeat a node, except that possibly $\mathsf{src}(p) = \mathsf{tgt}(p)$.

Additionally, given a set of paths P over a graph database G, we say that $p \in P$ is a SHORTEST path in P, if $\mathsf{len}(p) \leq \mathsf{len}(p')$ for each $p' \in P$. We use $\mathsf{Paths}(G)$ to denote the (possibly infinite) set of all paths in a graph database G.

Path Queries in GQL and SQL/PGQ. Following the GQL and SQL/PGQ [14] standards, we define *path queries* as expressions of the form

$$\mathsf{selector?}\ \mathsf{restrictor}\ (\mathtt{x}, \mathtt{regex}, \mathtt{y})$$

where selector? means that selector is optional and `regex` is a regular expression. Furthermore, `x` and `y` are variables (denoted as $?x, ?y, \ldots$) or nodes in the graph (denoted $v, v', n, n', \ldots$). *Selectors* and *restrictors* are used to specify which paths are to be returned. Their grammar is:

$$\begin{aligned} \text{selector} &: \text{ANY} \mid \text{ANY SHORTEST} \mid \text{ALL SHORTEST} \\ \text{restrictor} &: \text{WALK} \mid \text{TRAIL} \mid \text{SIMPLE} \mid \text{ACYCLIC} \end{aligned}$$

A selector-restrictor combination is also called a *path mode*. We have 16 path modes: 3 selectors times 4 restrictors, plus the 4 restrictors without selector. However, WALK needs to be preceded by a selector [14] to avoid the need to return infinitely many paths in the presence of cycles, which gives 15 modes.

Next, we formally define the semantics of path queries. Let G be a graph database and q be a path query of the form

$$\text{restrictor}\ (?x, \texttt{regex}, ?y)\ .$$

For now, we omit the optional selector part and assume that `x` and `y` are distinct variables $?x$ and $?y$ respectively. (We denote variables with question marks to distinguish them from nodes in G.) We use $\mathsf{Paths}(G, \text{restrictor})$ to denote the set of all paths in G that are valid according to restrictor. For example, $\mathsf{Paths}(G, \text{TRAIL})$ is the set of all trails in G. We then define the semantics of q over G, denoted $[\![q]\!]_G$, where $q = \text{restrictor}\ (?x, \texttt{regex}, ?y)$, as follows:

$$\begin{aligned} [\![\text{restrictor}\ (?x, \texttt{regex}, ?y)]\!]_G = \{(v, v', p) \mid\ & p \in \mathsf{Paths}(G, \text{restrictor}), \\ & \mathsf{src}(p) = v, \mathsf{tgt}(p) = v', \\ & \mathsf{lab}(p) \in \mathcal{L}(\texttt{regex})\ \}\ . \end{aligned}$$

Here $\mathcal{L}(\texttt{regex})$ denotes the language of `regex`. Intuitively, for a query "TRAIL $(?x, \texttt{regex}, ?y)$", the semantics returns all tuples (v, v', p) such that p is a TRAIL in our graph that connects v to v' and $\mathsf{lab}(p) \in \mathcal{L}(\texttt{regex})$. The semantics of selectors is defined on a case-by-case basis. Using q to denote the selector-free query $q = \text{restrictor}\ (?x, \texttt{regex}, ?y)$, we now have:

- $[\![\text{ANY}\ q]\!]_G$ returns, for each pair of nodes (v, v') such that $(v, v', p) \in [\![q]\!]_G$, for some p, a *single tuple* $(v, v', p_{out}) \in [\![q]\!]_G$.
- $[\![\text{ANY SHORTEST}\ q]\!]_G$ returns, for each pair (v, v') such that $(v, v', p) \in [\![q]\!]_G$, for some p, a *single tuple* $(v, v', p_{out}) \in [\![q]\!]_G$ such that p_{out} is SHORTEST among all paths p' with $(v, v', p') \in [\![q]\!]_G$.
- $[\![\text{ALL SHORTEST}\ q]\!]_G$ returns, for each pair (v, v') such that $(v, v', p) \in [\![q]\!]_G$ for some p, *all tuples* $(v, v', p_{out}) \in [\![q]\!]_G$ such that p_{out} is SHORTEST among all paths p' with $(v, v', p') \in [\![q]\!]_G$.

Notice that the SHORTEST modes only select shortest paths *grouped by endpoints*. That is, the selected paths do not need to be the shortest in the entire output, they only need to be the shortest matching paths from v to v'. Furthermore, the semantics of ANY and ANY SHORTEST is non-deterministic when

there are multiple (shortest) paths connecting a pair of nodes. We also remind that GQL and SQL/PGQ prohibit the WALK restrictor to be present without any selector attached to it, in order to ensure a finite result set.

Finally, if x is a node n in G, then the semantics is defined exactly as above, but only keep the answers (v, v', p) where $v = n$. (Analogously if y is a node in G.) As such, one can also use queries of the form ALL TRAILS $(n, \mathtt{a}^+, ?y)$.

3 Backbone of PathFinder and the WALK Semantics

In this section we describe the main idea behind PathFinder and show how it can be used to evaluate RPQs under the WALK semantics. That is, we show how to treat queries of the form

$$\text{selector WALK } (v, \texttt{regex}, ?x) \ .$$

Recall that the selector is obligatory in GQL, since the set of all walks can be infinite. For simplicity of exposition, we assume for now that the starting point of the RPQ is fixed (i.e., we start from a node v) and discuss the other cases for endpoints later. Our approach relies heavily on the product construction of automata [26,36]. We present its full construction here, but in practice it is important to construct the product *lazily*, i.e., only construct the part of it that is needed to evaluate the query.

Product Graph. Given a graph database $G = (V, E, \rho, \lambda)$ and an expression of the form $q = (v, \texttt{regex}, \texttt{?x})$, the *product graph* is constructed by first converting the regular expression regex into an equivalent non-deterministic finite automaton $(Q, \Sigma, \delta, q_0, F)$. It consists of a set of states Q, a finite alphabet of edge labels Σ, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the initial state q_0, and the set of final states F. The construction of the automaton (without ε-transitions) can be done in time linear in the size of regex [39]. The product graph $G_\times$ is then defined as the graph database $G_\times = (V_\times, E_\times, \rho_\times, \lambda_\times)$, where

- $V_\times = V \times Q$;
- $E_\times = \{(e, (q_1, a, q_2)) \in E \times \delta \mid \lambda(e) = a\}$;
- $\rho_\times(e, d) = ((x, q_1), (y, q_2))$ if:
  - $d = (q_1, a, q_2)$
  - $\lambda(e) = a$
  - $\rho(e) = (x, y)$; and
- $\lambda_\times((e, d)) = \lambda(e)$.

Each node of the form (u, q) in $G_\times$ corresponds to the node u in G and, furthermore, each path P of the form $(v, q_0), (v_1, q_1), \ldots, (v_n, q_n)$ in $G_\times$ corresponds to a path $p = v, v_1, \ldots, v_n$ in G that (a) has the same length as P and (b) brings the automaton from state q_0 to q_n. As such, when $q_n \in F$, then this path in G matches regex. In other words, all nodes v' that can be reached from v by a path that matches regex can be found by using standard graph search algorithms (e.g., BFS/DFS) on $G_\times$ starting in the node (v, q_0).

While this approach allows finding pairs (v, v') in answers to an RPQ, we show next how it can be used to also find paths. Our approach is rooted in this fairly standard construction in automata theory, but on the other hand, it has nice advantages: (a) the product graph can indeed be explored lazily *for all 15 path modes*, (b) paths can be always returned *on-the-fly*, allowing pipelined execution, and (c) the approach is highly efficient (Sect. 5). We point out in which cases subtleties such as *unambiguity of the automaton* need to be taken into account to achieve a correct algorithm.

3.1 ANY (SHORTEST) WALK

We first treat the WALK restrictor combined with selectors ANY and ANY SHORTEST, that is, queries of the form:

$$q = \text{ANY (SHORTEST)? WALK } (v, \texttt{regex}, ?x) \tag{1}$$

The idea is that, given a graph database G and a query q of the form (1), we can perform a classical graph search algorithm such as BFS or DFS starting at the node (v, q_0) of the product graph $G_\times$, built from the automaton $\mathcal{A}$ for `regex` and G. Since both BFS and DFS support reconstructing a single (shortest in the case of BFS) path to any reached node, we obtain the desired semantics for RPQs of the form (1). Query evaluation is presented in Algorithm 1. The algorithm is a straightforward adaptation of BFS and DFS on a lazily constructed $G_\times$ with modifications that eliminate multiple paths reaching the same node and multiple accepting runs in an automaton. We present the algorithm in detail since we extend the approach to support different semantics later.

The basic object we manipulate is a *search state*, i.e., a quadruple of the form $(n, q, e, prev)$, where n is the node of G we are currently exploring, q is the current state of $\mathcal{A}$, while e is the edge of G we used to reach n, and $prev$ is a pointer to the search state we used to reach (n, q) in $G_\times$. Intuitively, the (n, q)-part of the search state allows us to track the node of $G_\times$ we are traversing, while e, together with $prev$ allows to reconstruct the path from (v, q_0) that we used to reach (n, q). The algorithm uses four data structures: Open, which is a queue if we want to run BFS or a stack if we want to run DFS, a dictionary Visited, used to track already visited states to avoid infinite loops (searchable by the (n, q) component of search state), Solutions, which is a set containing (pointers to) search states in Visited that encode a solution path, and ReachedFinal, which ensures that no solution is returned twice.

The algorithm explores the product $G_\times$ of G and $\mathcal{A}$ using either BFS (Open is a queue) or DFS (Open is a stack), starting from (v, q_0). It starts by initializing the data structures and setting up the start node in $G_\times$ (lines 2–5). The main loop of line 6 is the classical BFS/DFS algorithm that pops an element $(n, q, e, prev)$ from Open (line 7) and starts exploring its neighbors in $G_\times$ (lines 11–14). When exploring $(n, q, e, prev)$, we scan all the transitions (q, a, q') of $\mathcal{A}$ that originate from q, and look for neighbors of n in G reachable by an a-labeled edge (line 11). Here, writing $(n', q', edge') \in \text{Neighbors}((n, q, edge, prev), G, \mathcal{A})$

simply means that $\rho(edge') = (n, n')$ in G, and that $(q, \lambda(edge'), q')$ is a transition of $\mathcal{A}$. If the pair (n', q') has not been visited yet, we add it to Visited and Open (lines 12–14), which allows it to be expanded later on in the algorithm. When popping from Open in line 7, we also check if q is a final state and that n has not been reached by a solution path yet (line 8). In this case we found a new solution; i.e., a WALK from v to n whose label is in the language of `regex`, so we add it to Solutions (line 10) and record it as reached (line 9). The ReachedFinal set is used to ensure that each solution is returned only once. (Since $\mathcal{A}$ is nondeterministic, it could happen that two different runs of $\mathcal{A}$ accept the same path, or that two different paths reach the same end node via a different end state.)

Algorithm 1 Evaluation of $query$ = ANY (SHORTEST)? WALK $(v, \texttt{regex}, ?x)$

```
1:  function AnyWalk(G, query)
2:      A ← Automaton(regex)                          ▷ q0 initial, F final states
3:      Open.init(); Visited.init(); ReachedFinal.init()
4:      startState ← (v, q0, null, ⊥)
5:      Visited.push(startState); Open.push(startState)
6:      while Open ≠ ∅ do
7:          current ← Open.pop()                      ▷ current = (n, q, edge, prev)
8:          if q ∈ F and n ∉ ReachedFinal then
9:              ReachedFinal.add(n)
10:             Solutions.add(current)
11:         for each (n', q', edge') ∈ Neighbors(current, G, A) do
12:             if (n', q', *, *) ∉ Visited then
13:                 newState ← (n', q', edge', current)
14:                 Visited.push(newState); Open.push(newState)
```

We note that Solutions stores the pointers to states in Visited, which define a solution path. So, for each tuple $(n, q, e, prev)$ in Solutions after running the algorithm, a path from v to n can be reconstructed using the $prev$ part of search states stored in Visited. Furthermore, solutions can be enumerated once the algorithm terminates, or returned as soon as they are detected (line 10). The latter approach allows for a pipelined execution of the algorithm (see [16] for more details). Using BFS guarantees that the returned path is indeed shortest.

Algorithm 1 is fairly simple, but shows in detail both how to manipulate the product graph in an efficient manner and how to handle arbitrary RPQs while finding a single (shortest) path between each pair of nodes in the result set. In addition, it eliminates duplicate results in the sense that, for every node n of G, it only computes a single tuple of the form $(n, q, e, prev)$ in Solutions if q is accepting. In the following sections we show how this approach can be extended for more complex path modes, starting with finding *all* shortest paths.

3.2 ALL SHORTEST WALKS

We next show how to evaluate queries of the form

$$q = \text{ALL SHORTEST WALK } (v, \texttt{regex}, ?x). \tag{2}$$

For this, we extend the BFS version of Algorithm 1 in order to find *all shortest paths* between pairs (v, v') of nodes, instead of a single path—see Algorithm 2. The intuition simple: to obtain all shortest paths, upon reaching v' from v by a path conforming to `regex` *for the first time*, BFS always does so using a shortest path. The length of this path can then be recorded (together with v'). When a new path reaches the same, already visited node v', if it has length equal to the recorded length for v', then this path is also an answer to our query. Interestingly, *the algorithm explores precisely the same portion of* $G_\times$ *as Algorithm* 1.

Algorithm 2 Evaluation of $query = \text{ALL SHORTEST WALK } (v, \texttt{regex}, ?x)$.

```
1:  function AllShortestWalk(G, query)
2:      A ← UnambiguousAutomaton(regex)            ▷ q0 initial, F final states
3:      Open.init(); Visited.init(); ReachedFinal.init()
4:      startState ← (v, q0, 0, ⊥)
5:      Visited.push(startState); Open.push(startState)
6:      while Open ≠ ∅ do
7:          current ← Open.pop()                   ▷ current = (n, q, depth, prevList)
8:          if q ∈ F then
9:              if n ∉ ReachedFinal then
10:                 ReachedFinal.add(⟨n, depth⟩)
11:                 Solutions.add(current)
12:             else if ReachedFinal.get(n).depth = depth then
13:                 Solutions.add(current)
14:         for next ← (n', q', edge') ∈ Neighbors(current, G, A) do
15:             if (n', q', *, *) ∈ Visited then
16:                 (n', q', depth', prevList') ← Visited.get(n', q')
17:                 if depth + 1 = depth' then     ▷ New shortest path to (n', q')
18:                     prevList'.add(⟨current, edge'⟩)
19:             else
20:                 prevList.init()
21:                 prevList.add(⟨current, edge'⟩)
22:                 newState ← (n', q', depth + 1, prevList)
23:                 Visited.push(newState); Open.push(newState)
```

As before, we use $\mathcal{A}$ to denote the NFA for `regex`. We will additionally require that $\mathcal{A}$ is *unambiguous* (it has at most one accepting run for every word), which we need to ensure that we do not return the same path twice. This condition is easy to enforce for real-world RPQs [9,10]. The main difference to Algorithm 1

is in the *search state* structure. A search state is now a quadruple of the form $(n, q, depth, prevList)$, where n is a node of G and q a state of $\mathcal{A}$, *depth* is the length of a shortest path to (n, q) from (v, q_0), while *prevlist* is a *list* of pointers to any previous search state that allows us to reach n via a shortest path from v. We assume that *prevList* is a *linked list*, initialized as empty, and accepting sequential insertions of pairs $\langle searchState, edge\rangle$ using add(). Intuitively, *prevList* allows us to reconstruct *all* shortest paths reaching a node.

When adding a pair $\langle searchState, edge\rangle$, we assume *searchState* to be a pointer to a previous search state, and *edge* will be used to reconstruct the path passing through the node in this previous search state. Finally, we again assume that Visited is a dictionary of search states, with the pair (n, q) being the search key. Namely, there can be at most one tuple $(n, q, depth, prevList)$ in Visited with the same pair (n, q). We assume that with Visited.get(n, q), we will obtain the unique search state having n and q as the first two elements.

Algorithm 2 explores the product graph of G and $\mathcal{A}$ using BFS, so Open is a queue. The main difference to Algorithm 1 is as follows: if a node (n', q') of the product graph $G_\times$ has already been visited (line 15), we do not directly discard the new path, but instead choose to keep it if and only if it is also shortest (line 17). In this case, the *prevList* for (n', q') is extended by adding the new path (line 18). If a new pair (n', q') is discovered for the first time, a fresh *prevList* is created (lines 20–23). As in Algorithm 1, we check for solutions after a state has been removed from Open (lines 8–13). Basically, when a state is popped from the queue, the structure of the BFS algorithm assures that we already explored all shortest paths to this state. *Notice that solutions can actually be returned before (i.e., after line 16 we can test if $q' \in F$).* Returning answers when popping from the queue in lines 8–13 has an added benefit that the paths are grouped for a pair (v, n) of connected nodes. As in Algorithm 1, we need to make sure that reaching the same node n via different accepting states of $\mathcal{A}$ is permitted only when this results in a shortest path. For this ReachedFinal is now a dictionary storing pairs $\langle n, depth\rangle$, where n is a solution node and *depth* the length of the shortest path reaching n from v. We assume n to be the dictionary key. In case that the solution is reached the first time (lines 9–11) we record this information. When reaching the same node with another accepting path (lines 12–13), we record the solution only if the length of the path is the same as the optimal length already recorded. Finally, we can enumerate the solutions by traversing the DAG stored in Visited in a depth-first manner starting from the nodes in Solutions. Next we illustrate how Algorithm 2 works in detail.

Example 1. Consider the social network graph G in Fig. 1 (a) and let

$$q = \text{ALL SHORTEST WALK } (\text{Joe}, \texttt{follows}^* \cdot \texttt{works}, ?x).$$

As we can see, there are three shortest paths connecting Joe to ENS Paris matching the query. To compute these, Algorithm 2 first converts the expression $\texttt{follows}^* \cdot \texttt{works}$ into the automaton in Fig. 1 (b). Algorithm 2 then starts traversing the product graph $G_\times$ from the node (Joe, q_0). Upon executing the algorithm, the structure of Visited is as follows:

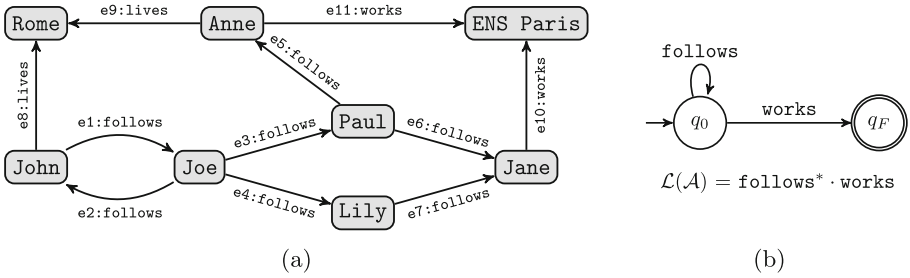
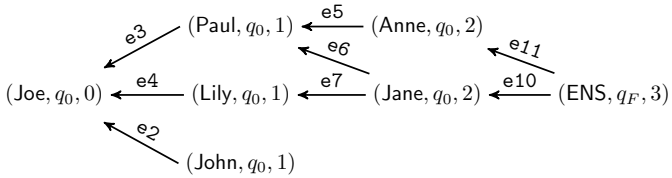


Fig. 1. A sample graph database (a) and a sample automaton $\mathcal{A}$ (b).



Here we represent *prevList* as a series of arrows to other states in Visited, and only draw $(n, q, depth)$ in each node. For instance, $(\mathsf{Jane}, q_0, 2)$ has two outgoing edges, representing two pointers in its *prevList*. The arrow is also annotated with the edge witnessing the connection (as stored in *prevList*).

To build this structure, Algorithm 2 explores the neighbors of (Joe, q_0); namely, $(\mathsf{Paul}, q_0), (\mathsf{Lily}, q_0)$ and (John, q_0) and puts them to Visited and Open, with $depth = 1$. The algorithm proceeds by visiting (Anne, q_0) from (Paul, q_0) and (Jane, q_0) from (Paul, q_0). The interesting thing happens in the next step when (Lily, q_0) is the node being expanded to its neighbor (Jane, q_0), which is already present in Visited. Here we trigger lines 15–18 of the algorithm for the first time, and update the *prevList* for (Jane, q_0), instead of ignoring this path. When we try to explore neighbors of (John, q_0), we try to revisit (Joe, q_0), so lines 15–18 are triggered again. This time the depth test in line 17 fails (we visited Joe with a length 0 path already), so this path is abandoned. We then explore the node (ENS, q_F) in $G_\times$ by traversing the neighbors of (Anne, q_0). Finally, (ENS, q_F) will be revisited as a neighbor of (Jane, q_0) on a previously unexplored shortest path. We can then enumerate all the paths from (ENS, q_F), by following the edges. □

For Algorithm 2 to work correctly, we crucially need $\mathcal{A}$ to be unambiguous. (this can be achieved by determinizing $\mathcal{A}$). Concerning run-time, Algorithm 2 indeed explores precisely the same portion of $G_\times$ as Algorithm 1, since the nodes of $G_\times$ we revisit (lines 15–18) do not get added to Open again. However, we can potentially add extra edges to Visited, so $[\![q]\!]_G$ can be much bigger (see [17,31] for examples when the set of shortest paths is exponential). Additionally, we note that when enumerating the output paths, each path is returned in time proportional to its length, which is also known as *output-linear delay* [19], and

is optimal in the sense that, to return a path we need to at least write it down. In short, we can conclude that Algorithm 2 runs with $O(|\mathcal{A}| \cdot |G|)$ pre-processing time, after which the results can be enumerated in time proportional to their total length.

4 TRAIL, SIMPLE, and ACYCLIC

We now devise algorithms for finding trails, simple paths, or acyclic paths. It is well established in the research literature that even checking whether there is a single path between two nodes that conforms to a regular expression and is a simple path or a trail is NP-complete [6,7,13,32], so we do not know any efficient algorithms. On the other hand, regular expressions for which this problem is indeed NP-hard are rare in practice [9,33,34] and, moreover, regular expressions for which these problems are tractable can be evaluated by cleverly enumerating paths in the product graph [33,35]. Our algorithms will follow this intuition and prune the search space whenever possible. In the worst case all such algorithms will be exponential but we will show that, on real-world graphs, the number of paths will be manageable, and the pruning technique will ensure that all dead-ends are discarded. We begin by showing how to evaluate queries of the form

$$q = (\mathsf{ALL\ SHORTEST})?\ \mathsf{restrictor}\ (v, \texttt{regex}, ?x)$$

where restrictor is TRAIL, SIMPLE, or ACYCLIC. Algorithm 3 shows how to evaluate such queries. Intuitively, the algorithm explores the product graph by enumerating all paths starting in (v, q_0) but pruning quickly. Here, our *search state* is a tuple $(n, q, depth, e, prev)$, where n is a node, q an automaton state, *depth* the length of a shortest path to (n, q) in $G_\times$, e an edge used to reach the node n, and *prev* a pointer to another search state stored in Visited, which is a set storing already visited search states. Same as in Algorithm 2, we use a dictionary ReachedFinal storing pairs $(n, depth)$, with key n. Here, n is a node reached in some query answer, and *depth* the length of a shortest path to this node.

The execution is very similar to Algorithm 1, but Visited is not used to discard solutions. Instead, when checking whether the current node can be extended to a neighbor (lines 18–21), we call IsValid which checks whether the path to the current node (i.e. n) in Visited can be extended to the next node (i.e. n') according to restrictor. Notice that we need to check whether the path in the *original graph* G satisfies the restrictor, and not the path in the product graph. If the explored neighbor allows to extend the current path according to restrictor, we add the new search state to Visited and Open (line 21). When popping from Open, we also check if a potential solution is reached (line 8). If the ALL SHORTEST selector is *not* present, we simply add the newly found solution. In the presence of the selector, we need to make sure to add only *shortest* paths to the solution set. The dictionary ReachedFinal is used to track already discovered nodes. If the node is seen for the first time, the dictionary is updated, and a new solution added (lines 11–13). Upon discovering the same node again (lines

Algorithm 3 Evaluation for $query =$ (ALL SHORTEST)? restrictor $(v, \texttt{regex}, ?x)$.

```
1: function AllRestrictedPaths(G, query)
2:     A ← UnambiguousAutomaton(regex)              ▷ q0 initial, F final states
3:     Open.init(); Visited.init(); ReachedFinal.init()
4:     startState ← (v, q0, 0, null, ⊥)
5:     Visited.push(startState); Open.push(startState)
6:     while Open ≠ ∅ do
7:         current ← Open.pop()                     ▷ current = (n, q, depth, e, prev)
8:         if q ∈ F then
9:             if !(ALL SHORTEST) then
10:                Solutions.add(current)
11:            else if n ∉ ReachedFinal then
12:                ReachedFinal.add(⟨n, depth⟩)
13:                Solutions.add(current)
14:            else
15:                optimal ← ReachedFinal.get(n).depth
16:                if depth = optimal then
17:                    Solutions.add(current)
18:        for next ← (n', q', edge') ∈ Neighbors(current, G, A) do
19:            if isValid(current, next, restrictor) then
20:                new ← (n', q', depth + 1, edge', current)
21:                Visited.push(new); Open.push(new)
```

14–17), a new solution is added only if it is shortest. Once all paths have been explored (Open $= \emptyset$), we can enumerate the solutions as in other algorithms.

The correctness of the algorithm crucially depends on the fact that $\mathcal{A}$ is unambiguous, since otherwise we could record the same solution twice. Termination is assured by the fact that eventually all paths that are valid according to the restrictor will be explored, and no new search states will be added to Open. Unfortunately, since we will potentially enumerate all the paths in the product graph, the complexity is $O\big((|\mathcal{A}| \cdot |G|)^{|G|}\big)$. This is also the best known runtime since the problem is NP-hard [13].

Adding ANY and ANY SHORTEST. Treating queries of the form

$$q = \text{ANY (SHORTEST)? restrictor } (v, \texttt{regex}, ?x)$$

can be done with minimal changes to Algorithm 3. Namely, we would use ReachedFinal as a *set* that stores the node first time a solution is found in order not to repeat any results. In addition, we would replace lines 8–17 with:

```
if q ∈ F and n ∉ ReachedFinal then
    ReachedFinal.add(n)
    Solutions.add(current)
```

Same as Algorithm 3, worst-case runtime is $O\big((|\mathcal{A}| \cdot |G|)^{|G|}\big)$, which is also the best known runtime due to NP-completeness of the problem [13].

5 Experimental Evaluation

We empirically evaluate PathFinder and show that the approach scales over real-world and synthetic datasets and queries. For code, data, and queries see [16].

Implementation. PathFinder is implemented as the path processing engine of MillenniumDB [48], a recent graph database system developed by the authors. MillenniumDB provides the infrastructure for processing GQL queries, generating execution plans and storing the data, while PathFinder executes path queries as described in the paper. The default data storage of MillenniumDB is B+trees. By default we assume all the data to be on disk and the main memory buffer to be empty. *All the algorithms are implemented in a fully pipelined fashion, meaning that they can be paused as soon as a result is detected.* While in the paper we assumed the source of the RPQ part of the query to be fixed, the implementation supports ends being fixed or free.

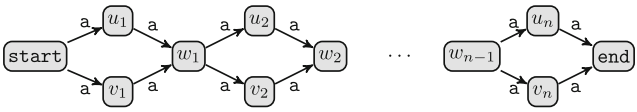


Fig. 2. Graph database with exponentially many paths.

Experimental Setup. We perform two sets of experiments:

(1) Wikidata. To test the efficiency and scalability of PathFinder we use Wikidata [47] and queries from its public SPARQL query log [30]. We use WDBench [2], a recently proposed Wikidata SPARQL benchmark. WDBench provides a curated version of the data set based on the truthy dump of Wikidata [50], which is an edge-labeled graph with 364 million nodes and 1.257 billion edges, using more than 8,000 different edge labels. The data set is publicly available [3]. WDBench provides multiple sets of queries extracted from Wikidata's public endpoint query log. We use the Paths query set, which contains 659 2RPQ patterns. From these, 592 have a fixed starting point or ending point (or both), while 67 have both endpoints free. We note that some queries require regular expressions which are not supported in all tested systems. The 659 patterns are then used in our tests under different GQL path modes (details below).

(2) Diamond. We test what happens when there is a large number of paths in our graph. The database is from [31] and presented in Fig. 2. The queries retrieve paths from start to end using a-labeled edges. All such paths are, at the same time, shortest, trails and simple paths, and have length $2n$. There are 2^n such paths, while the graph only has $3n + 1$ nodes and $4n$ edges. We test our query with the path modes from Sect. 2, while scaling n from 1 to 40.

Tested Systems. We use both BFS and DFS versions of PathFinder, denoted as PF-BFS and PF-DFS respectively. To compare with state of the art in the

area we tested Neo4J version 4.4.12 [49] (Neo4j for short); NebulaGraph version 3.5.0 [45] (Nebula); Kuzu version 0.0.6 [28] (Kuzu); Jena TDB version 4.1.0 [42] (Jena); Blazegraph version 2.1.6 [44] (BlazeGraph); and Virtuoso version 7.2.6 [15] (Virtuoso). Note that SPARQL engines do not support returning paths. For other systems, we test TRAIL and WALK modes.

How We Ran the Experiments. We used a machine with an Intel®Xeon® Silver 4110 CPU, and 128 GB of DDR4/2666 MHz RAM, running Linux Debian 10 with kernel v5.10. The hard disk for storing the data was a 14 TB HDD SEAGATE model ST14000NM001G. Custom indexes for speeding up the queries were created for Neo4j, Nebula and Kuzu and the systems were run with the default settings and no limit on RAM usage. Jena, BlazeGraph, Virtuoso and PathFinder were assigned 64 GB of RAM for buffering. Queries were executed in succession one after the other. *All queries were run with a limit of 100,000 results and a timeout of 1 min* (as suggested by WDBench [2]).

5.1 Performance Analysis over Wikidata

We managed to load the dataset into the three SPARQL engines and Neo4j. We did not manage to load the data into NebulaGraph [45], Kuzu [28] and Memgraph [43]. Results are summarized in Fig. 3. Next, we analyze the results.

WALKS. In Fig. 3 (left) we show the results for the WALK restrictor. We present box plots for the 659 queries. The first two columns represent the BFS and DFS version of Algorithm 1 in PathFinder, which corresponds to ANY (SHORTEST) WALK mode. Here, PathFinder clearly outperforms all other systems, *even if it returns paths whereas the SPARQL engines do not.* Indeed, Jena timed out 95 times, and BlazeGraph and Virtuoso 86 and 24 times, respectively, whereas PathFinder-BFS only had 8 and PathFinder-DFS 9 timeouts. Neo4j even timed out in 657 out of 659 queries in ANY SHORTEST WALK, so we do not include it in Fig. 3 (left). The final column shows the performance of PathFinder for the ALL SHORTEST PATH mode of Algorithm 2. Interestingly, *despite requiring a more involved algorithm, finding 100,000 paths under this path mode shows almost identical performance to finding a single shortest path for each reached node.* The number of timeouts here was only 7. Again, while Neo4j does support this path mode, it could only complete 2 out of 659 queries. Overall, we can conclude that PathFinder presents a stable strategy for finding WALKS and significantly outperforms the state-of-the-art.

TRAILS. Results for the TRAIL semantics are shown in Fig. 3 (right). The first two columns correspond to ANY SHORTEST TRAIL and ANY TRAIL in PathFinder. This performance is almost identical to the ANY WALK case, however with 25 and 24 timeouts for the BFS and DFS version, respectively. The following three columns correspond to the ALL TRAIL mode, supported by PathFinder-BFS, PathFinder-DFS, and Neo4j. We see an order of magnitude improvement in PathFinder compared to Neo4j and, additionally, only 9 and 10 timeouts for PathFinder-BFS and PathFinder-DFS,

versus 134 for Neo4j. The final column corresponds to ALL SHORTEST TRAIL in PathFinder, which again shows similar performance to other TRAIL-based modes, with 21 timeouts. Overall, all PathFinder variations show remarkably stable performance. Interestingly, while the TRAIL mode is evaluated using a brute-force approach, and is intractable theoretically [6], over real-world data it works remarkably well. This is most likely due to the fact that PathFinder can either detect 100,000 results rather fast, or because the data itself permits no further graph exploration. We also ran the experiments for SIMPLE and ACYCLIC restrictor in PathFinder, with identical results as in the TRAIL case, showing that Algorithm 3 is indeed a good option for real-world use cases.

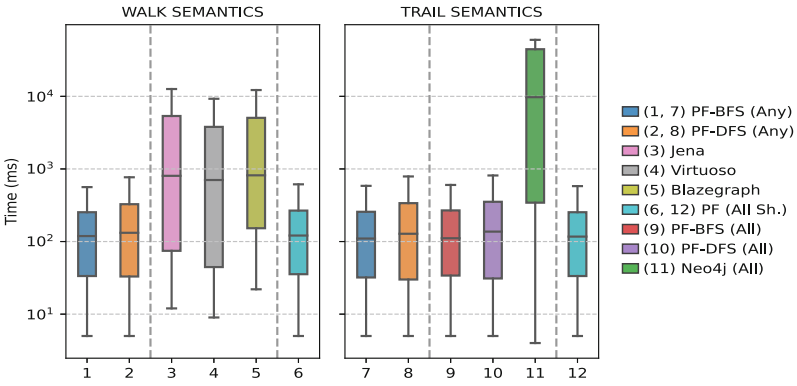


Fig. 3. Runtimes for the Wikidata experiment.

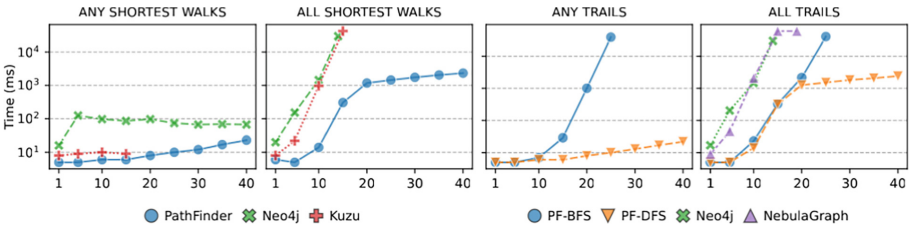


Fig. 4. WALK and TRAIL queries on the graph of Fig. 2.

Take-Home Message. Since all RPQ evaluation algorithms in the existing systems are based on traversing the parse tree of the regular expression [8,27,37,46], our results show that automata-based methods can even significantly outperform the regular expression based algorithms. It is even possible to return up to 100,000 paths for RPQ answers faster than it takes the SPARQL engines to only detect the existence of an answer.

5.2 Diamond: Scaling the Number of Paths

We test the performance of the query returning paths from start to end in the graph of Fig. 2. We run the experiment by setting $n = 1, \ldots, 40$, which allows to test scalability when the number of paths is large, i.e. 2^n. For each value of n we will look for the first 100,000 results. To compare with other engines, we focus on the WALK restrictor and the TRAIL restrictor. All the other paths modes in PathFinder, which is the only system supporting them, have identical performance as in the TRAIL case, since they are all derivatives of Algorithm 3. Since SPARQL systems cannot return paths, we exclude them from this experiment.

WALKS. The runtimes for ANY SHORTEST WALK and ALL SHORTEST WALK are in Fig. 4. We compare with Neo4j and Kuzu, since Nebula only supports TRAIL. Due to the small size of the graph, we run each query twice and report the second result. This is due to minuscule runtimes which get heavily affected by initial data loading. As we can observe, for ANY SHORTEST WALK (Fig. 4 (left)) all the engines perform well (we also pushed this experiment to $n = 1000$ with no issues). Kuzu only works up to $n = 15$ since the longest path it supports is of length 30. In the case of ALL SHORTEST WALK (Fig. 4 (2nd from left)), Neo4j times out for $n = 16$. Same as before, Kuzu stops at $n = 15$ with a successful execution. Overall, PathFinder seems to have a linear time curve in this experiment, while the other engines scale exponentially, showing the full power of Algorithm 2 when returning 100,000 paths.

TRAILS. Apart from comparing with other systems, this experiment allows to determine which traversal strategy (BFS or DFS) is better suited for Algorithm 3 in extreme cases such as the graph of Fig. 2. We present the results for ANY TRAIL and TRAIL in Fig. 4 (right). Concerning ANY TRAIL, which is only supported by PathFinder, the BFS-based algorithm will time out already for $n = 26$, which is to be expected, since it will construct all paths of length $1, 2, \ldots, 25$, before considering the first path of length 26. In contrast, DFS will find the required paths rather fast. When it comes to TRAIL mode, which retrieves *all* trails, the situation is similar. Here we also compare with other engines that find trails. As we can see, no engine apart from PathFinder-DFS could handle the entire query load as they all show an exponential performance curve. This illustrates that for a huge number of *trails*, DFS is the strategy of choice.

6 Related Work

Most work in the area focuses on finding nodes connected by an RPQ-conforming path and not on returning the paths [7,12,18,22,32,33,52]. Notable extensions include [23] where paths are returned, but not according to an RPQ pattern, and [41], which uses a BFS-style exploration to find the first k paths, meaning some non-shortest paths will be returned. Similar approach for top-k results is presented in [1], but not preferring shortest paths. Closest to our work is [31], where a compact representation of RPQ-conforming paths (called a PMR) is

presented for *some* GQL path modes. Compared to [31], we support *all* GQL path modes and present implementable algorithms, whereas [31] mostly focuses on theoretical guarantees. Interestingly, one can show that the Visited structure of our algorithms in fact encodes the PMR of [31] for any GQL path mode.

7 Conclusions

We present PathFinder, a unifying framework for returning paths in RPQ query answers. We believe PathFinder to be the first system that allows returning paths under *every* mode prescribed by the GQL and SQL/PGQ query standards [14], and that our experimental evaluation shows the approach to be highly competitive on realistic workloads. While our work was developed in the context of property graphs, it is straightforward to implement it on top of an existing SPARQL engine (see [16] for an example of this). We showed that returning paths that match RPQs can be practically viable, which opens the question to which extent we want to explore similar functionality for SPARQL [24].

Supplemental Material Statement. Code and experimental data is available at our repository [16], where we also provide an extended version of the paper.

Acknowledgments. Farías, Rojas and Vrgoč were supported by ANID – Millennium Science Initiative Program – Code ICN17_002. Vrgoč was also supported by the ANID Fondecyt Regular project 1240346. Martens was supported by ANR project EQUUS ANR-19-CE48-0019; funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), project number 431183758.

References

1. Aebeloe, C., Montoya, G., Setty, V., Hose, K.: Discovering diversified paths in knowledge bases. Proc. VLDB Endow. **11**(12), 2002–2005 (2018). https://doi.org/10.14778/3229863.3236245
2. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: WDBench: a Wikidata graph query benchmark. In: The Semantic Web - ISWC 2022 (2022). https://doi.org/10.1007/978-3-031-19433-7_41
3. Angles, R., Aranda, C.B., Hogan, A., Rojas, C., Vrgoč, D.: WDBench Dataset Download (2022). https://doi.org/10.6084/m9.figshare.19599589
4. Angles, R., et al.: G-CORE: a core for future graph query languages. In: International Conference on Management of Data (SIGMOD) (2018). https://doi.org/10.1145/3183713.3190654
5. Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J.L., Vrgoč, D.: Foundations of modern query languages for graph databases. ACM Comput. Surv. **50**(5) (2017). https://doi.org/10.1145/3104031
6. Baeza, P.B.: Querying graph databases. In: Symposium on Principles of Database Systems (PODS), pp. 175–188 (2013). https://doi.org/10.1145/2463664.2465216
7. Bagan, G., Bonifati, A., Groz, B.: A trichotomy for regular simple path queries on graphs. In: Symposium on Principles of Database Systems (PODS), pp. 261–272 (2013). https://doi.org/10.1145/2463664.2467795

8. Blazegraph source code (2024). https://github.com/blazegraph
9. Bonifati, A., Martens, W., Timm, T.: Navigating the maze of Wikidata query logs. In: The World Wide Web Conference (WWW), pp. 127–138. ACM (2019). https://doi.org/10.1145/3308558.3313472
10. Bonifati, A., Martens, W., Timm, T.: An analytical study of large SPARQL query logs. VLDB J. **29**(2–3), 655–679 (2020). https://doi.org/10.1007/s00778-019-00558-9
11. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Rewriting of regular expressions and regular path queries. J. Comput. Syst. Sci. **64**(3), 443–465 (2002). https://doi.org/10.1006/JCSS.2001.1805
12. Casel, K., Schmid, M.L.: Fine-grained complexity of regular path queries. In: International Conference on Database Theory (ICDT), pp. 19:1–19:20 (2021). https://doi.org/10.4230/LIPICS.ICDT.2021.19
13. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A graphical query language supporting recursion. In: International Conference on Management of Data (SIGMOD), pp. 323–330 (1987). https://doi.org/10.1145/38713.38749
14. Deutsch, A., et al.: Graph pattern matching in GQL and SQL/PGQ. In: International Conference on Management of Data (SIGMOD) (2022). https://doi.org/10.1145/3514221.3526057
15. Erling, O.: Virtuoso, a hybrid RDBMS/graph column store. IEEE Data Eng. Bull. **35**(1), 3–8 (2012). http://sites.computer.org/debull/A12mar/vicol.pdf
16. Farías, B., Martens, W., Rojas, C., Vrgoč, D.: PathFinder: a unified approach for handling paths in graph query languages (2024). https://github.com/AnonCSR/PathFinder
17. Farías, B., Rojas, C., Vrgoč, D.: MillenniumDB path query challenge. In: Alberto Mendelzon Workshop (AMW) (2023). https://ceur-ws.org/Vol-3409/paper13.pdf
18. Fionda, V., Pirrò, G., Gutiérrez, C.: Nautilod: a formal language for the web of data graph. ACM Trans. Web **9**(1), 5:1–5:43 (2015). https://doi.org/10.1145/2697393
19. Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., Vrgoc, D.: Efficient enumeration algorithms for regular document spanners. ACM Trans. Database Syst. **45**(1), 3:1–3:42 (2020). https://doi.org/10.1145/3351451
20. Francis, N., et al.: A researcher's digest of GQL (invited talk). In: International Conference on Database Theory (ICDT) (2023). https://doi.org/10.4230/LIPIcs.ICDT.2023.1
21. Francis, N., et al.: Cypher: an evolving query language for property graphs. In: International Conference on Management of Data (SIGMOD) (2018). https://doi.org/10.1145/3183713.3190657
22. Gubichev, A.: Query processing and optimization in graph databases. Ph.D. thesis, Technical University Munich (2015). https://nbn-resolving.org/urn:nbn:de:bvb:91-diss-20150625-1238730-1-7
23. Gubichev, A., Neumann, T.: Path query processing on very large RDF graphs. In: WebDB 2011 (2011). http://webdb2011.rutgers.edu/papers/Paper21/pathwebdb.pdf
24. Harris, S., Seaborne, A., Prud'hommeaux, E.: SPARQL 1.1 query language. W3C Recommendation (2013). https://www.w3.org/TR/sparql11-query/
25. Hogan, A., et al.: Knowledge graphs. ACM Comput. Surv. **54**(4), 71:1–71:37 (2022). https://doi.org/10.1145/3447772
26. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Boston (1979)
27. Apache jena source code (2024). https://github.com/apache/jena

28. Jin, G., Feng, X., Chen, Z., Liu, C., Salihoglu, S.: Kùzu graph database management system. In: 13th Conference on Innovative Data Systems Research. CIDR 2023, 8–11 January 2023, Amsterdam, The Netherlands (2023). www.cidrdb.org , https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf
29. Jumper, J., et al.: Highly accurate protein structure prediction with AlphaFold. Nature **596**(7873), 583–589 (2021). https://doi.org/10.1038/s41586-021-03819-2
30. Malyshev, S., Krötzsch, M., González, L., Gonsior, J., Bielefeldt, A.: Getting the most out of Wikidata: semantic technology usage in Wikipedia's knowledge graph. In: International Semantic Web Conference (ISWC) (2018). https://doi.org/10.1007/978-3-030-00668-6_23
31. Martens, W., Niewerth, M., Popp, T., Rojas, C., Vansummeren, S., Vrgoč, D.: Representing paths in graph database pattern matching. Proc. VLDB Endow. **16**(7), 1790–1803 (2023). https://www.vldb.org/pvldb/vol16/p1790-martens.pdf
32. Martens, W., Niewerth, M., Trautner, T.: A trichotomy for regular trail queries. In: International Symposium on Theoretical Aspects of Computer Science (STACS), pp. 7:1–7:16 (2020). https://doi.org/10.4230/LIPICS.STACS.2020.7
33. Martens, W., Trautner, T.: Evaluation and enumeration problems for regular path queries. In: International Conference on Database Theory (ICDT), pp. 19:1–19:21 (2018). https://doi.org/10.4230/LIPICS.ICDT.2018.19
34. Martens, W., Trautner, T.: Bridging theory and practice with query log analysis. SIGMOD Rec. **48**(1), 6–13 (2019). https://doi.org/10.1145/3371316.3371319
35. Martens, W., Trautner, T.: Dichotomies for evaluating simple regular path queries. ACM Trans. Database Syst. **44**(4), 16:1–16:46 (2019). https://doi.org/10.1145/3331446
36. Mendelzon, A.O., Wood, P.T.: Finding regular simple paths in graph databases. In: Very Large Data Bases, pp. 185–193 (1989). http://www.vldb.org/conf/1989/P185.PDF
37. Neo4j open source code (2024). https://github.com/neo4j/neo4j
38. Reutter, J.L., Soto, A., Vrgoč, D.: Recursion in SPARQL. Semant. Web **12**(5), 711–740 (2021). https://doi.org/10.3233/SW-200401
39. Sakarovitch, J.: Elements of Automata Theory. Cambridge University Press, Cambridge (2009)
40. Sakr, S., et al.: The future is big graphs: a community view on graph processing systems. Commun. ACM **64**(9), 62–71 (2021). https://doi.org/10.1145/3434642
41. Savenkov, V., Mehmood, Q., Umbrich, J., Polleres, A.: Counting to k or how SPARQL1.1 property paths can be extended to top-k path queries. In: SEMANTiCS 2017 (2017). https://doi.org/10.1145/3132218.3132239
42. Team, J.: Jena TDB (2021). https://jena.apache.org/documentation/tdb/
43. Team, M.: Memgraph (2023). https://memgraph.com/
44. Thompson, B.B., Personick, M., Cutcher, M.: The Bigdata® RDF graph database. In: Harth, A., Hose, K., Schenkel, R. (eds.) Linked Data Management, pp. 193–237. Chapman and Hall/CRC (2014). http://www.crcnetbase.com/doi/abs/10.1201/b16859-12
45. Vesoft Inc/Nebula: NebulaGraph (2023). https://www.nebula-graph.io/
46. Virtuoso open source code (2024). https://github.com/openlink/virtuoso-opensource
47. Vrandecic, D., Krötzsch, M.: Wikidata: a free collaborative knowledgebase. Commun. ACM **57**(10), 78–85 (2014). https://doi.org/10.1145/2629489
48. Vrgoč, D., et al.: MillenniumDB: an open-source graph database system. Data Intell. **5**(3), 560–610 (2023). https://doi.org/10.1162/dint_a_00229

49. Webber, J.: A programmatic introduction to Neo4j. In: SPLASH (2012). https://doi.org/10.1145/2384716.2384777
50. Wikidata:database download (2021). https://www.wikidata.org/wiki/Wikidata:Database_download
51. ten Wolde, D., Singh, T., Szárnyas, G., Boncz, P.A.: DuckPGQ: efficient property graph queries in an analytical RDBMS. In: Conference on Innovative Data Systems Research (CIDR) (2023). https://www.cidrdb.org/cidr2023/papers/p66-wolde.pdf
52. Yakovets, N., Godfrey, P., Gryz, J.: Query planning for evaluating SPARQL property paths. In: International Conference on Management of Data (SIGMOD), pp. 1875–1889. ACM (2016). https://doi.org/10.1145/2882903.2882944