Representing Paths in Graph Database Pattern Matching

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

Matthias Niewerth
University of Bayreuth
Germany
matthias.niewerth@uni-bayreuth.de

Tina Popp
University of Bayreuth
Germany
tina.popp@uni-bayreuth.de

Carlos Rojas
IMFD Chile
Chile
c.rojasvictoriano@gmail.com

Stijn Vansummeren
UHasselt, Data Science Institute
Belgium
stijn.vansummeren@uhasselt.be

Domagoj Vrgoč
PUC Chile & IMFD Chile
Chile
dvrgoc@ing.puc.cl

ABSTRACT

Modern graph database query languages such as GQL, SQL/PGQ, and their academic predecessor G-Core promote paths to first-class citizens in the sense that their pattern matching facility can return *paths*, as opposed to only nodes and edges. This is challenging for database engines, since graphs can have a large number of paths between a given node pair, which can cause huge intermediate results in query evaluation.

We introduce the concept of *path multiset representations (PMRs)*, which can represent multisets of paths exponentially succinctly and therefore bring significant advantages for representing intermediate results. We give a detailed theoretical analysis that shows that they are especially well-suited for representing results of regular path queries and extensions thereof involving counting, random sampling, and unions. Our experiments show that they drastically improve scalability for regular path query evaluation, with speedups of several orders of magnitude.

PVLDB Reference Format:
Wim Martens, Matthias Niewerth, Tina Popp, Carlos Rojas, Stijn Vansummeren, and Domagoj Vrgoč. Representing Paths in Graph Database Pattern Matching. PVLDB, 16(7): 1790 - 1803, 2023.
doi:10.14778/3587136.3587151

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/MillenniumDB/pmr.

1 INTRODUCTION

Graph databases are becoming increasingly popular [51]. Indeed, modern graph query languages such as Neo4j's Cypher [23], Tigergraph's GSQL [32], and Oracle's PGQL [50] are rapidly gaining adoption in industry, and there are ongoing ISO standardization efforts for GQL (a native query language for property graphs) as well as SQL/PGQ (which extends SQL with capabilities for graph pattern matching on property graphs) [24].

At the core of all of these languages lies the problem of evaluating *regular path queries* (or *RPQs* for short), which have been studied in database research since the late 1980s, see, e.g., [7, 10, 11, 16, 17, 22, 28, 43–45]. In essence, an RPQ consists of a regular expression e. The classical semantics of RPQs in the academic literature and in, e.g., implementations of SPARQL [63] is the following. When we evaluate e over an edge-labeled graph G, we return all node pairs (x, y) such that there exists a path from x to y in G whose sequence of edge labels forms a word in the language of e. Modern graph query languages such as GQL, SQL/PGQ, and their academic predecessors such as G-Core [4], are adopting a fundamentally different approach by making paths *first-class citizens*: RPQs no longer simply return endpoint pairs, but also the matching paths. We illustrate both semantics by means of the following example.

Example 1.1. We adopt the property graph of Figure 1 as our running example. The graph has *node identifiers* (a1, …, a6, c1, c2, p1,…,p4, ip1, ip2) in red and *edge identifiers* (t1, …, t8, li1, …, li6, hp1, …, hp6) in blue. Nodes and edges can carry *labels* (such as Account, Transfer, and isLocatedIn) and *property-value* pairs (such as (owner, Mike) and (date, 1/1/2020)). We depict labels and property/value pairs for nodes in solid boxes, whereas for edges, these are in dashed boxes (or in the legend on the bottom right).

Consider the RPQ consisting of the regular expression $e = \mathsf{Transfer}^+$. When evaluated under the classical semantics on the graph in Figure 1, this RPQ returns all node pairs (x, y) such that there is a path of length at least one from x to y in which every edge carries the label Transfer. Examples of such node pairs are (a1, a3) (which have a direct Transfer link) but also (a1, a2) (connected by a path of length 2) and (a1, a4) (connected by a path of length 3). When evaluated under the new semantics, this RPQ would return the matching paths in addition to the endpoint pairs, and include

(a1, a3, path(a1, t1, a3)),
(a1, a2, path(a1, t1, a3, t2, a2)),
(a1, a4, path(a1, t1, a3, t2, a2, t3, a4)),

where we used path(a1, t1, a3, t2, a2) to denote the path of length 2 from Scott (a1) to Aretha (a2), through Mike (a3). Under this semantics, RPQs hence return triples (x, y, p) where x and y are nodes and p is a path that connects them. □

Making paths a first-class citizen in modern graph query languages is not a straightforward task. Fundamentally, a key problem that systems are facing is how to best *represent* results of queries and subqueries that feature paths. The main issue is dealing with the sheer number of results that path queries can produce.

Proceedings of the VLDB Endowment, Vol. 16, No. 7 ISSN 2150-8097.
doi:10.14778/3587136.3587151

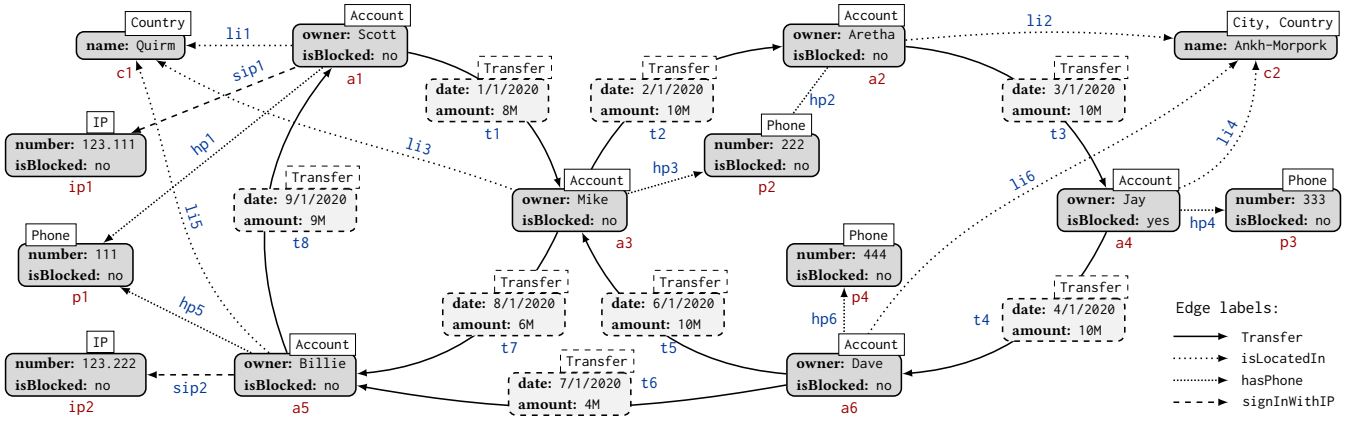


Figure 1: A property graph with information on bank accounts, their location, and financial transations, based on [24].

A common approach, proposed by the GQL standard [24], by SQL/PGQ [24], and already supported by multiple engines [31, 46, 55, 57, 62], is to return the results of a path query as a relational table. In Example 1.1, for instance, this table contains triples (x, y, p), where x and y are nodes and p is a Transfer-labeled path connecting them. We show a portion of this table (replacing node/edge IDs with their content for readability) in Table 1a. However, the number of such triples, and hence also the size of the table, can quickly become prohibitively large, or even infinite. To illustrate this, notice that the graph in Figure 1 has several Transfer-labeled cycles. This, in turn, implies that the number of triples (x, y, p) is infinite. For instance, there are infinitely many paths between a1 and a3, of lengths 1, 5, 9, etc. To ensure that queries have finite answers, the GQL language and today's query engines restrict the paths that are allowed. Common types of restrictions considered are: TRAIL (no repeated edge), SIMPLE (no repeated node), and SHORTEST [24].

While these restricted evaluation modes do fix the infinity issue, they can still result in prohibitively large outputs. To illustrate, consider the graph in Figure 2, which has $3n+1$ nodes and $4n$ edges. If we were to output all the shortest paths between x and y in Figure 2, there are 2^n of these. Notice that these paths are also both trails and simple paths. Therefore, a relational table representation of this output, such as the one in Table 1a and which current systems use, would "materialize" all 2^n paths. For this reason, it seems desirable to adopt a different data structure that can represent sets of triples (x, y, p) as succinctly as possible, preferably in less than 2^n space, while still allowing to generate the relational table representation from them.

We note that some systems already have something in place that can be seen as a succinct representation of paths, but we will explain why it is not sufficient. Query engines such as Neo4J [23] present query results to the user by means of so-called *graph projections*. Intuitively speaking, the graph projection takes the table representation and displays the subgraph of the original graph consisting only of the nodes and edges mentioned in the table. For instance, the graph projection of the query that asks for all paths from node x to node y in Figure 2 simply yields the graph of Figure 2 itself. In general, however, graph projections are not accurate representations of query results. Consider, for instance a second query that

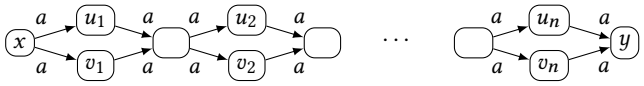


Figure 2: A graph with 2^n shortest paths from x to y.

returns only two paths in Figure 2: the path from x to y through $u_1, u_2, \ldots, u_n$ and the path from x to y through $v_1, v_2, \ldots, v_n$. The graph projection for this second query is *exactly* the same as that for the first query. So, graph projections are not lossless — they are just a subgraph of the input, and as such they lose the information about which paths were to be returned. Furthermore, to the best of our knowledge, current systems compute the graph projection *from the tabular output*. This is inefficient since the latter can be exponentially larger than both the graph projection and the input.

Our Contribution

In this paper we introduce the concept of *path multiset representations (PMRs)* for compactly representing (multi)sets of paths. We show that processing of (generalized) RPQs based on PMRs can make query evaluation drastically more efficient. In a nutshell, PMRs aim to combine the best of the relational table representation and graph projections while avoiding their disadvantages. That is, they provide a *compact* representation of an exponential (or even infinite) number of paths, similarly to graph projections, while at the same time being *lossless* and allowing to identify individual paths in the output, as the tabular representation does.

Our formal and experimental results show significant potential:

(1) PMRs can represent sets and multisets of paths accurately and exponentially more succinctly compared to current state-of-the-art systems and the current GQL standard description.
(2) PMRs can represent the output of *regular path queries*, the basic building block of modern graph pattern matching languages, and can be computed in linear time combined complexity, which strongly contrasts to the current exponential algorithms.
(3) On PMRs we can perform operations that are common on the tabular representation: enumeration (i.e., scanning), counting, random sampling, grouping, and taking unions. By performing

Table 1: Tabular representation of `Transfer`-trails in Figure 1.

(a) Tabular representation.

x	y	p
Mike	Billie	Mike -[Transfer]-> Billie
Billie	Scott	Billie -[Transfer]-> Scott
Mike	Aretha	Mike -[Transfer]-> Aretha
[...]		
Mike	Aretha	Mike -[Transfer]-> Billie -[Transfer]-> Scott -[Transfer]-> Mike -[Transfer]-> Aretha
Mike	Billie	Mike -[Transfer]-> Billie -[Transfer]-> Scott -[Transfer]-> Mike -[Transfer]-> Aretha -[Transfer]-> Jay -[Transfer]-> Dave -[Transfer]-> Billie
[...]		

(b) Pairwise grouped tabular representation.

x	y	p
Mike	Billie	Mike -[Transfer]-> Billie, Mike -[Transfer]-> Billie -[Transfer]-> Scott -[Transfer]-> Mike -[Transfer]-> Aretha -[Transfer]-> Jay -[Transfer]-> Dave -[Transfer]-> Billie, [...]
Mike	Aretha	Mike -[Transfer]-> Aretha, Mike -[Transfer]-> Billie -[Transfer]-> Scott -[Transfer]-> Mike -[Transfer]-> Aretha [...]
[...]		

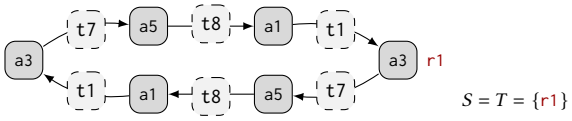


Figure 3: Path representation over the graph database of Figure 1, representing all cycles of even length from Mike to Mike where the transferred amounts are strictly less than 10M. We depicted the value $\gamma(u)$ inside every node u.

these operations on PMRs instead of on tabular representations, we hence obtain query plans that are exponentially more efficient in general because we can avoid computing the exponentially larger tabular representation.

(4) As a proof of concept, we experimentally evaluate queries with RPQs and aggregation on synthetic and real-world data. We find that *today's systems easily time out for queries that return paths on large-scale data* and we show that *PMRs have a significant potential to improve this situation, yielding speedups of up to four orders of magnitude.*

The presented theoretical results show that exponential speed-ups in query evaluation methods are possible, while our experiments confirm speed-ups of several orders of magnitude on small and large data sets. In the wider context of query language design, PMRs show that it is possible to represent infinitely many paths in query evaluation plans using a finite object, which opens up further possibilities for the design of future graph query languages.

PMRs in a Nutshell. Intuitively, a PMR over a graph G is itself a graph R, together with

- a homomorphism γ from R to G, and
- a set of "start nodes" S and a set of "target nodes" T.

The idea is that R provides a succinct structure to represent paths between groups of nodes in G.

To illustrate, Figure 3 shows a PMR R over the graph G of Figure 1. It uses a single start node, `r1`, which is also the single target node, and represents all cycles of even length from Mike to Mike where the transferred amounts are less than 10M. Intuitively, the homomorphism γ associates each node in R to a node in G — for each node u of R, we depicted the value of $\gamma(u)$ inside the node u in Figure 3. Notice that γ can associate multiple nodes in R to the same node in G. In particular, the leftmost and rightmost node in R are both mapped to `a3`. This symbolizes the fact that one needs to traverse the cycle `a3`–`a5`–`a1`–`a3` twice to obtain even length.

A PMR R "represents" a (possibly infinite) number of paths in G. These paths are the images of the paths in R from some node in S to some node in T under the mapping γ. As such, in Figure 3, the paths from $S = \{\texttt{r1}\}$ to $T = \{\texttt{r1}\}$ are cycles of length 0, 6, 12, etc. in R, which correspond (through γ) to cycles of the same lengths in G. In this case, the number of paths represented by R is infinite.

An example PMR representing exponentially many paths would be the graph in Fig. 2 with $S = \{x\}$, $T = \{y\}$, and γ the identity.

Paper Organization. We provide mathematical background in Section 2. In Section 3 we define Path Multiset Representations and study their basic properties. In Section 4 we introduce (unions of) Generalized Regular Path Queries (GRPQs) as a formal model of classical Regular Path Queries that also return paths. In Section 5 we show how to evaluate (U)GRPQs using Path Multiset Representations. We experimentally evaluate PMRs in Section 6. We discuss related work in Section 7 and conclude in Section 8. Because of space limitations, some details and proofs are omitted. An extended version of this paper, whose appendix contains those items is available online [42]. This version also discusses the equivalence and minimization problems for PMRs.

2 PRELIMINARIES

Background. For a natural number n, we denote the set $\{1, \ldots, n\}$ by $[n]$. A *multiset* M is a function from a set S to $\mathbb{N} \setminus \{0\} \cup \{\infty\}$. We denote multisets using double braces, e.g., in the multiset $M = \{\!\{a, a, b\}\!\}$, we have that $M(a) = 2$ and $M(b) = 1$. We do not distinguish between sets and multisets where all elements have multiplicity one: i.e., we equate $\{\!\{a, b\}\!\} = \{a, b\}$. For a multiset M we denote by $\mathrm{set}(M)$ the set obtained from M by forgetting multiplicities. For instance, $\mathrm{set}(\{\!\{a, a, b\}\!\}) = \{a, b\} = \{\!\{a, b\}\!\}$.

Graph databases. We assume that we have infinite *disjoint* sets NID of *node identifiers*, EID of *edge identifiers*, and $\mathbf{L}$ of *labels*.

Because our focus in this paper will be on how *paths as first-class citizens* interact with regular path queries on graph databases, we adopt a formal data model that is a simplified version of property graphs in which property graph features that are non-essential to our discussion, such as node labels and property-value records, are omitted. We stress that this is only for ease of exposition: all of these features can be added to our approach without influencing

our results. Formally, our data model is an edge-labeled directed multigraph, defined as follows.

Definition 2.1. A *graph database* is a tuple $G = (N, E, \eta, \lambda)$, where

(1) $N \subseteq \mathsf{NID}$ is a finite set of node identifiers and $E \subseteq \mathsf{EID}$ is a finite set of edge identifiers;
(2) $\eta : E \to (N \times N)$ is a total function, called the *incidence mapping*, that associates each edge to the nodes it connects;
(3) $\lambda : E \to \mathbf{L}$ is a total function, called the *labeling function*, that associates a label to each edge.

In what follows, if G is a graph then we will write N_G for the set of G's nodes, and similarly write E_G, η_G, λ_G for the set of G's edges, incidence mapping, and labeling function. We may omit subscripts if G is clear from the context.

An *unlabeled graph* is a triple (N, E, η) defined exactly as a graph database, except that the labeling function λ is missing.

Paths. A *path* in a graph database G is a sequence

$$\rho = v_0 e_1 v_1 e_2 v_2 \cdots e_n v_n$$

with $n \geq 0$, $e_i \in E$, and $\eta(e_i) = (v_{i-1}, v_i)$ for every $i \in [n]$. We sometimes write $\mathsf{path}(\rho)$ instead of simply ρ to stress that we are talking about a path. For example $\mathsf{path}(\mathtt{a1}, \mathtt{t1}, \mathtt{a3}, \mathtt{t2}, \mathtt{a2})$ is the path of length two from Scott to Aretha in Figure 1. We use $\mathsf{Paths}(G)$ to denote the set of paths in G.

If ρ is a path in G and λ is G's labeling function, then we write $\lambda(\rho)$ for the sequence of edge labels $\lambda(\rho) = \lambda(e_1) \cdots \lambda(e_n)$ occurring on the edges of ρ. We write $\mathsf{src}(\rho)$ for the node v_0 at which ρ starts, and $\mathsf{tgt}(\rho)$ for the node v_n at which it ends. Given two sets of nodes S and T, we say that ρ is a path *from S to T* if $\mathsf{src}(\rho) = v_0 \in S$ and $\mathsf{tgt}(\rho) = v_n \in T$.

A *path multiset* over G (or PM over G for short) is a multiset of paths, all in the same graph G. We will often simply speak about path multisets without referring to the graph that they are drawn from, which will be implicit from the context.

3 PATH MULTISET REPRESENTATIONS

To the best of our knowledge, intermediate or final results of queries in current graph database query languages such as Cypher [23], G-Core [4], and SQL-PGQ [24] are always represented as tables in which each path is listed explicitly, essentially as in Table 1a. Our focus is on representing the *path multisets* involved in query answers in a drastically more succinct manner.

Example 3.1. Consider the set of all paths from x to y in Figure 2. Since there are 2^n such paths, representing them as in Table 1 would take 2^n rows. Instead, we next propose to represent this set of paths by means of the graph in Figure 2 itself, together with the set $\{x\}$ of source nodes and $\{y\}$ of target nodes. This representation has size $O(n)$ instead of $\Omega(2^n)$.

More precisely, we propose to use *path multiset representations of G*, which we define next.

Definition 3.2. A *path multiset representation (PMR) over graph G* is a tuple $R = (N, E, \eta, \gamma, S, T)$, where

(1) (N, E, η) is an unlabeled graph;
(2) $\gamma : (N \cup E) \to (N_G \cup E_G)$ is a (total) homomorphism, i.e. a function that maps nodes in R to nodes in G and edges in R to edges in G such that, if an edge $e \in E$ connects v_1 to v_2 in R, then $\gamma(e)$ connects $\gamma(v_1)$ to $\gamma(v_2)$ in G; and
(3) $S, T \subseteq N$ are sets of *source* and *target* nodes, respectively.

If R is a PMR, then we sometimes write N_R for its set of nodes, and similarly E_R, η_R, γ_R, S_R, and T_R for the other components. If R_1 and R_2 are PMRs over the same graph G whose nodes and edges are disjoint, then we write $R_1 \sqcup R_2$ for the PMR over G obtained by taking the disjoint union of R_1 and R_2 (defined in the obvious way by taking the union of each component).

If R is a PMR of G, we say that node $v \in N$ *represents* the node $\gamma(v)$ in G. Furthermore, each path

$$\rho = v_0 e_1 v_1 e_2 v_2 \cdots e_n v_n$$

from S to T in R *represents* a path in G, namely the path

$$\gamma(\rho) := \gamma(v_0)\gamma(e_1)\gamma(v_1)\gamma(e_2)\gamma(v_2) \cdots \gamma(e_n)\gamma(v_n) \ .$$

We define $\mathsf{SPaths}(R)$ and $\mathsf{MPaths}(R)$ to be the set, resp. multiset, of paths represented by R, that is,

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

A PMR R *represents* a multiset M of paths if $M = \mathsf{MPaths}(R)$. It represents a set of paths $P = \{\rho_1, \rho_2, \ldots\}$ if $P = \mathsf{SPaths}(R)$. Notice that, if $M = \mathsf{MPaths}(R)$, then we always have that $\mathsf{set}(M) = \mathsf{SPaths}(R)$. In other words, if a PMR represents a multiset of paths, it also represents the corresponding set of paths.

A PMR R is *trim* if every node in N_R is on some path from some node in S to some node in T. As such, trim PMRs do not contain useless information. Our interest will be in constructing trim PMRs.

3.1 Examples of PMRs

If γ is the identity function, then a PMR is structurally a subgraph of G. This is already useful, as we illustrated in Example 3.1. By choosing a different γ, however, we can incorporate *state information*, which is necessary for evaluating regular path queries (Example 3.3), and *multiplicities* of paths (Example 3.4).

Example 3.3 (State information). Figure 3 shows a PMR R for all cycles of even length from Mike to Mike, and where all transferred amounts are strictly less than 10M. (We omitted node and edge IDs that are irrelevant.) The "even length" condition can be encoded in R, since γ can map different nodes in R to the same node in G.

Example 3.3 illustrates another interesting property of PMRs: they can represent an infinite number of paths in a finite manner. Indeed, the set of cycles of even length from Mike to Mike in Example 3.3 is infinite. We have cycles of length 6, 12, 18, etc.

Example 3.4 (Multisets). The PMR R in Figure 4 represents the path of length two from Mike to Scott twice. We have that $\mathsf{SPaths}(R) = \{\mathsf{path}(\mathtt{a3}, \mathtt{t7}, \mathtt{a5}, \mathtt{t8}, \mathtt{a1})\}$ and $\mathsf{MPaths}(R) = \{\!\{\mathsf{path}(\mathtt{a3}, \mathtt{t7}, \mathtt{a5}, \mathtt{t8}, \mathtt{a1}), \mathsf{path}(\mathtt{a3}, \mathtt{t7}, \mathtt{a5}, \mathtt{t8}, \mathtt{a1})\}\!\}$.

To represent results of queries (or intermediate results in query plans), our aim is to work with PMRs R such that $\mathsf{MPaths}(R)$ corresponds to the multiset semantics of the query. We discuss how this is done for regular path queries in Section 4.

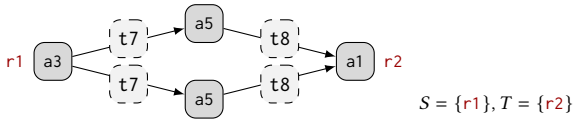


Figure 4: Path representation over the graph database of Figure 1, representing the path of length two from Mike to Scott twice. We depicted the value $\gamma(u)$ inside every node u.

3.2 Basic Properties of PMRs

We make some easy but important observations about the path multisets that can be represented by PMRs. Let G be a graph database.

Any single path. Every single path in G can be represented by a PMR. Specifically, for a path $\rho = v_0 e_1 v_1 e_2 v_2 \cdots e_n v_n$ in G, define its *canonical PMR* R_ρ as $R_\rho = (N, E, \eta, \gamma, S, T)$ where

$$\begin{array}{lll} N = \{\mathbf{v}_0, \ldots, \mathbf{v}_n\} & \eta(\mathbf{e}_i) = (\mathbf{v}_{i-1}, \mathbf{v}_i) & \text{for all } i \in [n] \\ E = \{\mathbf{e}_1, \ldots, \mathbf{e}_n\} & \gamma(\mathbf{e}_i) = \mathbf{e}_i & \text{for all } i \in [n] \\ S = \{\mathbf{v}_0\},\ T = \{\mathbf{v}_n\} & \gamma(\mathbf{v}_j) = v_j & \text{for all } j \in [0, n] \end{array}$$

Then $\mathsf{MPaths}(R_\rho) = \mathsf{SPaths}(R_\rho) = \{\rho\}$. Notice that we construct all nodes $\mathbf{v}_i$ and edges $\mathbf{e}_i$ in the PMR to be pairwise distinct, while this is not necessarily the case for the nodes v_i and edges e_i in ρ (e.g., when ρ has loops). This pairwise distinctness is necessary to ensure that $\mathsf{MPaths}(R_\rho)$ and $\mathsf{SPaths}(R_\rho)$ are exactly the singleton $\{\rho\}$: had we simply taken R_ρ to be the subgraph of G induced by ρ then, if ρ contains loops, both $\mathsf{MPaths}(R_\rho)$ and $\mathsf{SPaths}(R_\rho)$ would be infinite instead of the desired singleton.

Any finite multiset of paths. Let $M = \{\!\{\rho_1, \ldots, \rho_k\}\!\}$ be a finite multiset of paths in G. For each path ρ_i, let R_i be the canonical PMR of ρ_i and assume w.l.o.g. that the sets of nodes and edges of these representations are pairwise disjoint. Define the *canonical PMR* R_M of M to be the disjoint union $R_1 \sqcup \cdots \sqcup R_n$ of the individual canonical PMRs. Then $\mathsf{MPaths}(R_M) = M$. In the special case where every path in M occurs only once, and M is hence a set of paths, then we also have that $\mathsf{SPaths}(R_M) = M$.

Proposition 3.5. *Let M be a finite multiset of paths in G. Then there exists a path representation of G that represents M.*

While any finite multiset M of paths can hence always be represented by means of the canonical *PMR*, this representation is not necessarily the smallest possible PMR for M. In the extended version of this paper [42], we therefore give insight into the complexity of the equivalence and minimization problems for PMRs.

The reader may wonder about which infinite multisets of paths in G can be represented by PMRs. It turns out that these are precisely the regular multisets, i.e., the multisets M such that $\mathrm{set}(M)$ is a regular language, i.e., there exists an NFA A such that $L(A) = \mathrm{set}(M)$ and $M(\rho)$ is the number of accepting runs of A on ρ, for every path ρ. Notice that the alphabet of A is $N_G \cup E_G$.

4 GENERALIZED REGULAR PATH QUERIES

Regular path queries are a crucial feature that sets graph query languages apart from relational query languages, since they allow us to easily ask queries about arbitrarily long paths in graphs. Furthermore, they are central in Cypher [23], SQL/PGQ, and GQL [24]. Although regular path queries have been studied in research for decades (e.g., [10, 16, 22, 44, 45]), their incarnation in Cypher, SQL/PGQ, and GQL is different: they now have the capability of returning entire paths instead of just their endpoints. In this section, we introduce *generalized regular path queries* (GRPQs) to formalize this important extra feature.

Regular languages, expressions, and automata. We recap some basics on regular expressions and regular languages. A set of words (each word using symbols from our fixed set of labels **L**) is also called a *language*. A *regular expression* is an expression of the form

$$\mathsf{exp} ::= \varepsilon \mid a \mid \mathsf{exp}_1 \mathsf{exp}_2 \mid \mathsf{exp}_1 + \mathsf{exp}_2 \mid \mathsf{exp}^*.$$

Here, ε denotes the empty word and a ranges over symbols in **L**. The *language* $\mathrm{L}(\mathsf{exp})$ of expression exp is defined as usual [33]. A language L is *regular* if there exists a regular expression exp such that $L = \mathrm{L}(\mathsf{exp})$.[1] Regular languages can equivalently be represented by finite state automata. We assume basic familiarity with deterministic (DFA) and non-deterministic finite automata (NFAs) [33], and omit their formal definition. We say that an NFA A is *unambiguous* (UFA for short) if it has at most one accepting run for every word. Every DFA is unambiguous, but the converse is not necessarily true. In what follows we will range over regular expressions by the meta-variable exp and over UFAs by the meta-variable ufa. We write $\mathrm{L}(\mathsf{exp})$ and $\mathrm{L}(\mathsf{ufa})$ to denote the language of exp and ufa, respectively.

Generalized regular path queries. While classical RPQs are syntactically defined to be simply a relational-calculus-like atom (x, L, y) of endpoint variables (x, y) and regular language L, we find it convenient for our purposes to develop GRPQs as a small algebraic query language. Specifically, our syntax for GRPQs completely ignores binding endpoints to endpoint variables, as this feature is unimportant for the immediate results that follow.

Formally, a *Generalized Regular Path Query* (GRPQ) is an expression φ of the form

$$\begin{array}{l} \varphi ::= L \mid \sigma_{U,V}(\varphi) \mid m(\varphi) \\ m ::= \mathsf{shortest}, \mathsf{simple}, \mathsf{trail} \end{array}$$

Here, L is regular language (possibly specified by a regular expression or UFA), U and V are either a finite set of node identifiers or the infinite set of all node identifiers,[2] and m is a *selector mode*. We will refer to sets of node identifiers like U and V that are either finite or the set of all nodes as *node predicates*.

Intuitively, L selects all paths that match L, whereas $\sigma_{U,V}$ restricts results to those for which the source and target endpoints belong to U and V, respectively, and m restricts results to those paths that are shortest, simple, or trail. Formally, a GRPQ φ, when evaluated on a graph database G, evaluates to a path multiset $\varphi(G)$ over G, inductively defined as follows. Let $\mathsf{Paths}(G)$ denote the (possibly

[1] Notice that we have expressions for all regular languages, except the emtpy language, which is typically not used in the context of RPQs.

[2] Our main use of $\sigma_{U,V}(\varphi)$ will be to restrict the endpoints of a result of a subquery φ to sets of nodes U and V that we have already computed elsewhere in the query plan. From a systems perspective, it helps to think of U and V as pointers to sets (or unary predicates on nodes) rather than the sets themselves.

infinite) set of all paths of G.

$$L(G) = \{\!\{\rho \in \mathsf{Paths}(G) \mid \lambda_G(\rho) \in L\}\!\},$$
$$\sigma_{U,V}(\varphi)(G) = \{\!\{\rho \in \varphi(G) \mid \mathsf{src}(\rho) \in U, \mathsf{tgt}(\rho) \in V\}\!\},$$
$$m(\varphi)(G) = m(\varphi(G)).$$

Let M be any multiset of paths of G. In the last line, the semantics of selector mode m is defined by

$$\mathsf{shortest}(M) = \{\rho \in M \mid \rho \text{ is a shortest path}\},$$
$$\mathsf{simple}(M) = \{\rho \in M \mid \rho \text{ is simple}\}, \text{ and}$$
$$\mathsf{trail}(M) = \{\rho \in M \mid \rho \text{ is a trail}\},$$

where a path ρ is a *shortest path*, if there exists no shorter path from $\mathsf{src}(\rho)$ to $\mathsf{tgt}(\rho)$ in M, it is *simple*, if each node appears at most once in ρ, and it is a *trail*, if each edge occurs at most once in ρ. Notice that $\mathsf{shortest}(M)$ can contain paths of different length, since we only remove paths ρ from M for which there are shorter paths from $\mathsf{src}(\rho)$ to $\mathsf{tgt}(\rho)$.

The operations supported in GRPQs correspond to path evaluation modes in the upcoming GQL standard [24] and the ones studied in the research literature. Specifically, the unrestricted version L corresponds to regular path queries [22], and can return an infinite amount of paths, such as in Example 1.1. The trail mode is supported by Cypher [47] and GQL [24]. The simple mode is similar, but reverses the role of nodes and edges, and has been studied in the literature [7, 44, 45]. Finally, shortest is supported by many existing systems [47, 55, 62], and the GQL standard [24]. For a theoretical study of shortest see [61].

Unions of GRPQs. Note in particular that $\varphi(G)$, as defined above, is actually a set of paths (no paths occur multiple times). This changes once we consider unions of GRPQs. A *union of GRPQs* (UGRPQ for short) is an expression given by the syntax

$$\psi ::= \varphi \mid \psi \uplus \psi,$$

where φ ranges over GRPQs, and $\uplus$ denotes multiset union. Formally, the semantics of a GRPQ ψ on a graph database G is given by $(\psi_1\ \psi_2)(G) = \psi_1(G) \uplus \psi_2(G)$. The multiplicity of a path in $(\psi_1 \uplus \psi_2)(G)$ is hence the sum of its multiplicity in $\psi_1(G)$ plus its multiplicity in $\psi_2(G)$.

Whenever convenient, in what follows, we will apply the operators of UGRPQs directly on path multisets. For example, for a PM M we write $\sigma_{U,V}(M)$ for $\{\!\{\rho \in M \mid \mathsf{src}(\rho) \in U, \mathsf{tgt}(\rho) \in V\}\!\}$.

Grouped output of GRPQs. A (U)GRPQ hence computes a path (multi)set. Note that the elements of a PM are unsorted, so there does not need to be any relationship between one path and the next. Sometimes, however, it is desirable for efficiency reasons to *group* the elements of a PM, on their source node, target node, or both. This is the case, for instance, when we wish to answer aggregate queries such as "compute, for each source node, the number of paths originating in that node", or "compute, for each pair of endpoints (u, v) the number of paths between them". We next formalize the notion of grouped path multisets.

Definition 4.1. A *(source/target/pairwise) grouped path multiset* (GPM) over a graph G is a partition H of a path multiset M into maximal multisets, such that the following condition is satisfied for each multiset $M' \in H$:

- *source grouped*: for all $\rho, \rho' \in M'$: $\mathsf{src}(\rho) = \mathsf{src}(\rho')$.
- *target grouped*: for all $\rho, \rho' \in M'$: $\mathsf{tgt}(\rho) = \mathsf{tgt}(\rho')$.
- *pairwise grouped*: for all $\rho, \rho' \in M'$: $\mathsf{src}(\rho) = \mathsf{src}(\rho')$ and $\mathsf{tgt}(\rho) = \mathsf{tgt}(\rho')$.

Notice that, if H is source grouped, it is a collection of multisets such that, for each $\rho_1 \in M_1 \in H$ and $\rho_2 \in M_2 \in H$ with $M_1 \neq M_2$, then $\mathsf{src}(\rho_1) \neq \mathsf{src}(\rho_2)$. (The other cases are analogous.)

Let M be a PM over a graph G. We define the following grouping operators on M, which return a source grouped, target grouped, and pairwise grouped GPM, respectively.

$$\mathsf{grp}_{\mathsf{src}}(M) = \{\sigma_{\{\mathsf{src}(\rho)\},N_G}(M) \mid \rho \in M\},$$
$$\mathsf{grp}_{\mathsf{tgt}}(M) = \{\sigma_{N_G,\{\mathsf{tgt}(\rho)\}}(M) \mid \rho \in M\},$$
$$\mathsf{grp}_{\mathsf{src,tgt}}(M) = \{\sigma_{\{\mathsf{src}(\rho)\},\{\mathsf{tgt}(\rho)\}}(M) \mid \rho \in M\}.$$

We refer to Figure 5 for a visualization of the different groupings. (The figure illustrates how we can use PMRs for representing the different groups, but may be helpful here nevertheless.)

We also introduce grouping at the query language level, and define a *grouped* UGRPQ to be an expression of the form $\mathsf{grp}_S(\psi)$ with ψ a UGRPQ and S a non-empty subset of $\{\mathsf{src}, \mathsf{tgt}\}$. The semantics of grouped GRPQs is the obvious one: $\mathsf{grp}_S(\psi)(G) = \mathsf{grp}_S(\psi(G))$.

Tabular output of (grouped) UGRPQs. A (U)GRPQ hence computes a path (multi)set, and a grouped UGRPQ computes a grouped path multiset.

GQL, SQL/PGQ, and Cypher represent path multisets by means of a relational table such as the one illustrated in Table 1a. To refer to this representation, for a PM M, we write $\mathsf{tab}(M)$ for the table containing the tuples $(\mathsf{src}(\rho), \mathsf{tgt}(\rho), \rho)$ for each $\rho \in M$. As such,

$$\mathsf{tab}(\psi(G)) = \{\!\{(\mathsf{src}(\rho), \mathsf{tgt}(\rho), \rho) \mid \rho \in \psi(G)\}\!\}.$$

We introduce a similar relational table representation on grouped PMs, and define

$$\mathsf{tab}(\mathsf{grp}_{\mathsf{src}}(M)) = \{(\mathsf{src}(M'), M') \mid M' \in \mathsf{grp}_{\mathsf{src}}(M))\}$$
$$\mathsf{tab}(\mathsf{grp}_{\mathsf{tgt}}(M)) = \{(\mathsf{tgt}(M'), M') \mid M' \in \mathsf{grp}_{\mathsf{tgt}}(M)\}$$
$$\mathsf{tab}(\mathsf{grp}_{\mathsf{src,tgt}}(M)) = \{\big(\mathsf{src}(M'), \mathsf{tgt}(M'), M'\big) \mid M' \in \mathsf{grp}_{\mathsf{src,tgt}}(M)\}$$

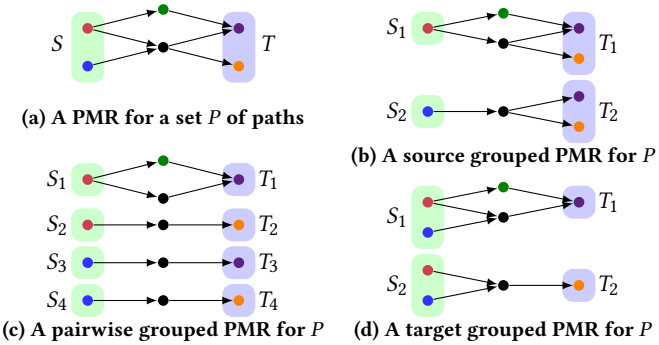
Here, we write $\mathsf{src}(M')$ (resp. $\mathsf{tgt}(M')$) for the unique source node (resp. target node) shared by all paths in M'.

It is important to stress the difference between $\mathsf{tab}(\psi(G))$ and $\mathsf{tab}(\mathsf{grp}_{\mathsf{src,tgt}}(\psi(G)))$: the former has one tuple per path in $\psi(G)$, while the latter has one tuple per group in $\mathsf{grp}_{\mathsf{src,tgt}}(\psi(G))$; the third component of that latter tuple is itself a path multiset. To illustrate, Table 1b shows the pairwise-grouped tabular representation for the Transfer-trails of Figure 1, while Table 1a shows the ungrouped tabular representation.

PMR output for (grouped) UGRPQs. Our interest in this paper is in using PMRs for succinctly representing the outputs of UGRPQs. In this respect, we say that a PMR R represents the output of UGRPQ ψ on graph G if it represents $\psi(G)$.

Similarly to how PMRs represent PMs, we introduce *grouped PMRs* to represent grouped PMs. Concretely, a *grouped PMR* is a finite set $S = \{R_1, \ldots, R_k\}$ of PMRs, such that $\mathsf{MPaths}(R_i)$ and $\mathsf{MPaths}(R_j)$ are disjoint, for every $i \neq j$. A grouped PMR *represents* a grouped PM H if $H = \{\mathsf{MPaths}(R_1), \ldots, \mathsf{MPaths}(R_k)\}$.

Figure 5 contains a PMR of five paths and illustrates different groupings of the set of paths. We use six different colors to show the six different nodes in G under the image of γ.



(a) A PMR for a set P of paths

(b) A source grouped PMR for P

(c) A pairwise grouped PMR for P

(d) A target grouped PMR for P

Figure 5: Grouped PMRs for the same set of paths.

5 ANSWERING UGRPQS

In Section 5.1 we explore how to compute (grouped) PMRs to represent the result of (grouped) UGRPQs. Subsequently, we discuss how to obtain the tabular ouput of (grouped) UGRPQs from PMRs in Section 5.2, where we also explore related problems such as counting the number of paths in a PMR, and drawing finite samples.

Model of computation. To analyze the complexity of our algorithms, we assume a RAM model of computation where the space used by node and edge ids, as well as integers, the time of arithmetic operations on integers, and the time of memory lookups are all $O(1)$. We further assume that hash tables have $O(1)$ access and update times while requiring linear space. While it is well-known that real hash table access is $O(1)$ *expected* time and updates are $O(1)$ *amortized* time, complexity results that we establish for this simpler model can be expected to translate to average (amortized) complexity in real-life implementations [21].

Throughout the rest of the paper, we assume an adjacency-list representation of graph databases and PMRs. As such, given a node u, it takes $O(1)$ time to retrieve the list of outgoing edges, while given an edge, it takes $O(1)$ time to retrieve its endpoints. Retrieving the label of an edge is also $O(1)$, and the same holds for retrieving the value of the homomorphism γ for a node or edge in a PMR.

5.1 Computing Path Multiset Representations

We first show how to compute a PMR for $\varphi(G)$ when φ is a regular language L. We will focus on the case where L is given as an unambiguous automaton ufa. In practice, regular languages are always given as a regular expression and, in theory, an exponential blow-up may occur when converting a regular expression to a UFA. However, we inspected the regular expressions in the query logs of [14, 15], with over 558 million SPARQL queries for Wikidata and DBpedia, containing 55 million RPQs, and we noticed that for none of these expressions such a blow-up actually occurs: the conversion is linear-time, even to a DFA. Our focus on UFAs is hence reasonable. In what follows, if φ is a GRPQ, we write $\varphi = \text{ufa}$ to indicate that φ is of the form L, where the language L is given by ufa.

The fundamental notion that underlies our construction for representing $\varphi(G)$ when $\varphi = \text{ufa}$ is the *product* between a graph database and ufa, which is defined as follows.

Definition 5.1 (Graph product). Assume given an unabmiguous automaton ufa $= (Q, \Sigma, \Delta, I, F)$, where Q is the set of UFA states, $\Sigma \subseteq \mathbf{L}$ is its set of used labels, $\Delta \subseteq Q \times \Sigma \times Q$ the set of transitions[3], $I \subseteq Q$ is the set of initial states, and $F \subseteq Q$ the set of final states. Let $G = (N_G, E_G, \eta_G, \lambda_G)$ be a graph database. Then the *product of G and* ufa, denoted as $G \times \text{ufa}$, is the PM representation over G defined as

- $N = N_G \times Q$
- $E = \{(e, (q_1, a, q_2)) \in E_G \times \Delta \mid a = \lambda_G(e)\}$
- $\eta((e, d)) = ((v_1, q_1), (v_2, q_2))$ such that
  - e is from v_1 to v_2 in G and
  - $d = (q_1, a, q_2)$, where $a = \lambda_G(e)$,
- $\gamma((v, q)) = v$, $\gamma((e, d)) = e$,
- $S = N_G \times I$, and
- $T = N_G \times F$.

We will denote by $\text{trim}(G \times \text{ufa})$ the subgraph of $G \times \text{ufa}$ that is obtained by removing all nodes and edges that do not participate in a path from S to T in $G \times \text{ufa}$. As such, $\text{trim}(G \times \text{ufa})$ is a trim path multiset representation.

Trimmed graph products provide a convenient way to obtain PM representations for GRPQs of the form $\varphi = \text{ufa}$. Indeed, we can show that $\text{trim}(G \times \text{ufa})$ represents the set $\varphi(G) = \{\!\{\rho \in \text{Paths}(G) \mid \lambda_G(\rho) \in \text{L}(\text{ufa})\}\!\}$ of all ufa-matched paths in G.

Theorem 5.2. *Let G be a graph database and let $\varphi = \text{ufa}$ be a GRPQ. Then both $G \times \text{ufa}$ and $\text{trim}(G \times \text{ufa})$ are PMRs of $\varphi(G)$, computable in linear time combined complexity $O(|\varphi||G|)$.*

We illustrate by means of the following example that the unambiguous property of ufa in Theorem 5.2 is important for the correctness of the construction. Specifically, it is needed to ensure correct multiplicities of paths.

Example 5.3. Consider the regular expressions $\text{exp}_1 = \text{Transfer} \cdot \text{Transfer}$ and $\text{exp}_2 = (\text{Transfer} \cdot \text{Transfer}) + (\text{Transfer} \cdot \text{Transfer})$. Notice that $L(\text{exp}_1) = L(\text{exp}_2)$ and that exp_2 is written in a "non-optimal" way. Nondeterministic automata that correspond to exp_1 and exp_2 are depicted in Figure 7: the left one is unambiguous (even deterministic) while the right one is not. Figure 6 illustrates a part of $\text{trim}(G \times \text{ufa})$ where G is the graph from Figure 1 and ufa is the left automaton from Figure 7, namely the part that is reachable from the node $(\text{a6}, 1)$. The resulting PMR represents three paths of length two in G, which means that three paths that match exp_1 start from a6 in Figure 1. Notice that, if we would apply the same construction using the right NFA of Figure 7, the result would have two additional nodes $(\text{a5}, 2')$ and $(\text{a3}, 2')$, leading to 6 paths in G (two copies of each path represented in Figure 6), which is incorrect.

Obviously, both constructions are correct if multiplicities are not important (i.e., we are interested in the SPaths semantics), but only the construction using the UFA has the correct multiplicities. □

Selection. We now consider GRPQs that involve the selection operator $\sigma_{U,V}$. Concretely, for a GRPQ $\varphi = \sigma_{U,V}(\text{ufa})$ and graph database G we can obtain a PMR of $\varphi(G)$ by constructing $G \times \text{ufa}$, but trimming differently. In general, we observe that a more general way of trimming allows us to express $\sigma_{U,V}$ on arbitrary PMRs.

[3] Without loss of generality, we do not use ε-transitions.

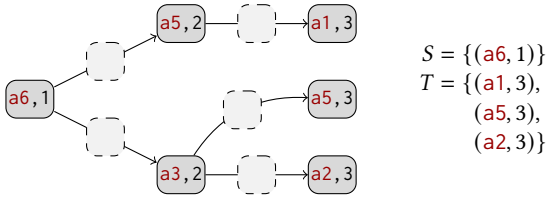


Figure 6: Illustration of the product construction..

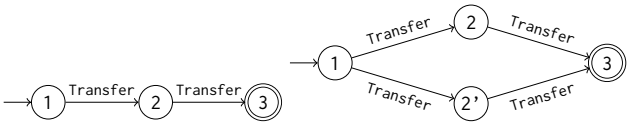


Figure 7: Two automata for the language Transfer·Transfer.

Concretely, let $R = (N, E, \eta, \gamma, S_R, T_R)$ be an arbitrary PMR over a graph G and let U, V be node predicates. Denote by $\mathrm{trim}(R, U, V)$ the subgraph of R that is obtained by removing all nodes and edges in R that do not participate in a path from $S' := \{u \in S_R \mid \gamma(u) \in U\}$ to $T' := \{u \in T_R \mid \gamma(u) \in V\}$. We can show that $\mathrm{trim}(R, U, V)$ represents $\sigma_{U,V}(\mathsf{MPaths}(R))$, leading to the following theorem. For a node predicate U, let $|U|$ denote the cardinality of U if U is finite, and let it be 1 if U is the infinite set of all node identifiers[4].

Theorem 5.4. *Let R be a PMR on graph G. Let U and V be node predicates. Then $\mathrm{trim}(R, U, V)$ is a PMR of $\sigma_{U,V}(\mathsf{MPaths}(R))$, computable in linear time $O(|R| + |U| + |V|)$.*

Consequently, we can evaluate GRPQs such as $\varphi = \sigma_{U,V}(\mathsf{ufa})$ simply by computing $\mathrm{trim}(G \times \mathsf{ufa}, U, V)$ which, by Theorems 5.2 and 5.4, can be done in linear time combined complexity $O(|\varphi||G| + |U| + |V|)$.

Grouping. Since source grouped, target grouped, and pairwise grouped representations of a PM M can always be obtained by repeatedly computing $\sigma_{U,V}(M)$ for different sets U and V, we obtain the following corollary from Theorem 5.4.

Corollary 5.5. *Let M be a PM on a graph G, represented by trim PMR R. Assume that X is the set of all source nodes in M, i.e., $X = \{\mathrm{src}(\rho) \mid \rho \in M\}$. Let Y be the set of all target nodes in M, and let XY be the set of all $(\mathrm{src}, \mathrm{tgt})$ pairs of paths in M. We can then compute*

(1) a grouped representation of $\mathsf{grp}_{\mathrm{src,tgt}}(M)$ in time $O(|XY||R|)$;
(2) a grouped representation of $\mathsf{grp}_{\mathrm{src}}(M)$ in time $O(|X||R|)$;
(3) a grouped representation of $\mathsf{grp}_{\mathrm{tgt}}(M)$ in time $O(|Y||R|)$.

We stress that the complexities given by Corollary 5.5 are attractive and, in a sense, optimal. Indeed, consider, for example, source grouping $\mathsf{grp}_{\mathrm{src}}(M)$. There are $|X|$ groups in the resulting grouped PM, and we hence need to represent every group by a PMR in a corresponding grouped PMR. Corollary 5.5 tells us that a representation for each such group can be obtained in linear time in the size of the original representation R of M. Since, without special preprocessing, we cannot even read R in less time, the resulting complexity is optimal.

[4]If U is the set of all node identifiers, it corresponds to the predicate 'True', which is concisely represented in constant space, hence we set $|U| = 1$ in this case.

Shortest paths. We next turn our attention to evaluating GRPQs that involve selector modes $m \in \{\mathrm{shortest}, \mathrm{simple}, \mathrm{trail}\}$. The next theorem shows that it is possible to apply shortest as operation on PMRs with favorable complexities. We will see later that $m = \mathrm{simple}$ and $m = \mathrm{trail}$ are more complex.

Theorem 5.6. *Let M be a PM over graph database G and let R be a trim PMR representing M. Let $k = \min(|S|, |T|)$ with S and T the sets of source and target nodes of R, respectively. From R we can compute a trim PMR for $\mathsf{shortest}(M)$ in time $O(k|R|)$.*

Simple paths and trails. We next turn to simple paths and trails. We start by noting that, if P ≠ NP, then there does not even exist a polynomial time algorithm for deciding if there *exists* a simple path or trail that matches a given regular expression between two given nodes [7, 43, 45]. This already implies the following:

Observation 5.7. *(a) Assume $\varphi = m(L)$ with $m \in \{\mathrm{simple}, \mathrm{trail}\}$. If from φ and graph G we can compute a PMR (or tabular representation) for $\varphi(G)$ in polynomial time, then P = NP.*
(b) Assume a PMR R for PM M and $m \in \{\mathrm{simple}, \mathrm{trail}\}$. If from R we can compute a PMR for $m(M)$ in polynomial time, then P = NP.

Within exponential time, however, we can minimize PMRs.

Proposition 5.8. *Given a PMR R for a path multiset M over G, and selector mode $m \in \{\mathrm{simple}, \mathrm{trail}\}$, we can compute from R a minimal PMR for $m(M)$ in exponential time.*

Although the complexity in this proposition is high, it is indeed unavoidable by Observation 5.7 and, furthermore, already the worst-case *size* of the set of paths represented in R is exponential.

Multiset unions. The fundamental problem when using PMRs for computing the result of a union of GRPQs is to compute a path representation for the multiset union of two PMRs. In our setup, this is very easy to do, as it suffices to take the component-wise union of the two PMRs. Therefore, evaluating UGRPQs is no more costly than evaluating the base queries.

Proposition 5.9. *Let R_1 and R_2 be two PMRs. We can compute a PMR for $\mathsf{MPaths}(R_1) \uplus \mathsf{MPaths}(R_2)$ in linear time $O(|R_1| + |R_2|)$.*

Conclusion and discussion. It directly follows from the results in this section (notably, Theorems 5.2, 5.4 and 5.6) that any UGRPQ ψ in which the regular languages are given as unambiguous automata and which uses only the selector mode shortest can be evaluated in *linear time combined complexity* when using PMRs to represent query outputs. This is in strong contrast to what today's systems do, since they compute $\mathrm{tab}(\psi(G))$ instead, which, as illustrated in the Introduction, is exponentially large in $|G|$ in general, even for $m = \mathrm{shortest}$. We note that our linear time combined complexity holds *even when* $\psi(G)$ is infinite. Furthermore, grouping on such UGRPQs can also be done efficiently by Corollary 5.5, proportional to the number of groups to be formed. The simple and trail selector modes are more complex to evaluate on PMRs, but we stress that this complexity is caused by the fundamental complexity of finding simple paths or trails that match a regular language. Also the tabular representation faces the same complexity.

Of course, simply representing UGRPQ outputs by means of PMRs may not be sufficient, since often we want to be able to retrieve (a part of) the tabular representation, or count the number of paths retrieved. We show next that PMRs fully support this.

5.2 Computing Output From PMRs

In this section we show that, from a given representation R of PM M we can efficiently generate the tabular representation $\mathsf{tab}(M)$ when M is finite, as well as compute the number of paths in M or draw a random sample.

Enumeration with output-linear delay. We wish to be careful with what we mean by "efficiently generate" $\mathsf{tab}(M)$ from R when M is finite. Indeed, because R can be exponentially more succinct than M and $\mathsf{tab}(M)$, the total time to generate $\mathsf{tab}(M)$ from R will obviously be exponential in R in the worst case. This exponential complexity is only due to the exponential number of tuples that we need to generate: we will show that generating individual tuples in $\mathsf{tab}(M)$ from R is efficient, in the sense that it takes only time proportional to the size of the tuple being generated—independently of the size of R, or M, or $\mathsf{tab}(M)$. To formalize this notion, we adopt the framework of *enumeration algorithms*. Enumeration algorithms are an attractive way of gauging the complexity of algorithms that need to generate large (or infinite) sets, which have recently received significant attention in the database community, both from a theoretical [2, 12, 29, 40, 52, 53] and practical point [34, 35, 59].

We require the following definitions. Given an input x, an algorithm is said to *enumerate* a multiset O if it outputs the elements of O one by one in some order $o_1, o_2, o_3, \ldots$, such that the number of times an element o occurs in this enumeration equals its multiplicity in O. (Repeated elements need not be subsequent in the enumeration.) In particular, if O is a set, then the enumeration cannot contain duplicates. It enumerates O *with output-linear delay* if the time required to output the i-th element o_i, measured as the difference in time between outputting o_{i-1} (or the start of the algorithm, when $i = 1$) and finishing outputting o_i, is proportional to the size of o_i, independent of the size of O or of the input x. If O is finite, then it is also required that the algorithm terminates immediately after outputting the last element. In that case, the total time that the algorithm takes to enumerate O is hence $O(|O|)$, i.e., linear in O.

Proposition 5.10. *Let M be a finite path multiset on a graph G.*

(1) From a trim PMR R of M we can enumerate both M and $\mathsf{tab}(M)$ with output-linear delay.
(2) From a trim grouped PMR R of $\mathsf{grp}_S(M)$ we can enumerate both $\mathsf{grp}_S(M)$ and $\mathsf{tab}(\mathsf{grp}_S(M))$ with output-linear delay, for any non-empty $S \subseteq \{\mathsf{src}, \mathsf{tgt}\}$.

Together with the results of Subsection 5.1, Proposition 5.10 tells us that we can evaluate an UGRPQ φ, which uses either the shortest selector, or no selector at all, on a graph database G, by running a polynomial preprocessing phase for computing the PMR for $\varphi(G)$, and then enumerating the results one-by-one in time that is proportional to the length of the output path. In a sense, one could argue that such enumeration is optimal, since this is the time it takes to write own the output. For trail and simple the same guarantee on enumeration holds, but constructing the appropriate PMRs now requires an exponential preprocessing phase.

We conclude this subsection by observing how PMRs can be used to count the number of query results, or sample paths in a GRPQ output uniformly at random. The former kind of result is relevant for dealing with queries that involve aggregation and the latter can be useful to provide uniform sampling guarantees to GQL's ANY-mode [24], if desired.

Proposition 5.11. *Let R be a trim PMR. Then we can*

(1) count the number of paths in $\mathsf{MPaths}(R)$ in linear time, where the returned result is $+\infty$ if $\mathsf{MPaths}(R)$ is infinite;
(2) if $\mathsf{MPaths}(R)$ is a finite multiset, uniformly at random sample a path in $\mathsf{MPaths}(R)$ in linear time.
(3) given a natural number $n \in \mathbb{N}$, uniformly at random sample a path from the submultiset of all paths of length n in $\mathsf{MPaths}(R)$, in time $O(n|R|)$.

6 EXPERIMENTS

Implementation. To empirically validate the potential of our approach, we use MillenniumDB [62], a recent open source graph database which stores the graph data on disk using B+trees. MillenniumDB already uses the product construction for evaluating RPQs, and we extend this capability to implement PMRs. For simplicity, our implementation focuses on GRPQs with a fixed start node, i.e., GRPQs φ of the form $\sigma_{\{s\},\mathsf{NID}}(L)$, with s a node id. It supports the following query modes. Let $M = \varphi(G)$ be the path set returned by the evaluation of φ on G.

(M1) Endpoints: return $\{\mathsf{tgt}(\rho) \mid \rho \in M\}$, i.e., the set of all nodes reachable by φ from s. This coincides with the non-generalized, i.e., standard semantics of RPQs. We will refer to the elements in this set as the endpoints of φ.
(M2) Single-Shortest: returning for each endpoint t a single pair (t, ρ) with ρ a shortest path from s to t.
(M3) All-Shortest: returning the set $\{(\mathsf{tgt}(\rho), \rho) \mid \rho \in \mathsf{shortest}(M)\}$ of all shortest paths;
(M4) Count: returning for each endpoint t the pair (t, c) with c the number of shortest paths from s to t; and
(M5) Shortest-PMR: constructing the PMR for $\mathsf{shortest}(M)$, representing all shortest paths (without enumerating them).

Evaluation under the endpoint mode is done by constructing the trim product graph (Section 5.1). The endpoints can be computed from this graph R by computing $\{\gamma_R(n) \mid n \in R_T\}$. For (M5) we have a dedicated physical operator that builds the PMR $\mathsf{shortest}(\mathsf{trim}(G \times \varphi))$, as described by Theorem 5.6, directly from G and φ, without first separately computing $\mathsf{trim}(G \times \varphi)$. This operator also allows to output a single shortest path for each endpoint, as required by (M2), and we use Proposition 5.10 to enumerate all shortest paths from $\mathsf{shortest}(\mathsf{trim}(G \times \varphi))$ for (M3). Finally, the counting algorithm is implemented as described in Proposition 5.11. The counting results are grouped by target, for the single source specified by the query. A LIMIT operator may be applied to query modes (M1)–(M3), in which case our implementation stops construction of the PMR when sufficiently many endpoints were found.

While all of our algorithms operate in main memory, the input graph is always loaded from disk via the MilleniumDB system buffer. Our implementation, together with the experiments and datasets in this section can be found at https://github.com/MillenniumDB/pmr.

Competitors and system setup. PMR refers to our implementation using PMRs. We compare to Neo4J version 4.4.12 (Neo for short), Jena TDB version 4.1.0 [54] (Jena), Blazegraph version 2.1.6 [56]

Table 2: Runtimes on WD

System	Evaluation mode	Timeouts	Average	Median
Neo	Endpoints (M1)	92	18.36s	8.11s
Jena	Endpoints (M1)	30	6.58s	0.43s
Blaze	Endpoints (M1)	40	8.27s	0.69s
Virtuoso	Endpoints (M1)	16	3.66s	0.54s
pmr	Endpoints (M1)	7	1.97s	0.16s
pmr	Single shortest (M2)	7	2.2s	0.16s
pmr	All shortest (M3)	5	2.2s	0.17s
pmr	Count (M4)	22	4.5s	0.08s
pmr	Construct (M5)	22	5.6s	0.07s

Table 3: Runtime of Neo, the SPARQL engines, and pmr on Q1, and Neo on Q1' (T/O denotes timeout)

Q1	Neo	Virtuoso	Blaze	Jena	pmr	Neo / Q1'
DMND	T/O	T/O	2,699 ms	415 ms	28 ms	17,373 ms
FB	T/O	T/O	2,586 ms	675 ms	130 ms	7,243 ms

(Blaze), and Virtuoso version 7.2.6 [26] (Virtuoso). All experiments were run on a commodity server with an Intel®Xeon®Silver 4110 CPU, and 128GB of DDR4/2666MHz RAM, running Linux Debian 10 with the kernel version 5.10. The hard disk used to store the data was a SEAGATE ST14000NM001G with 14TB of storage. Neo was used with default setting and no limit on RAM usage. Jena and Blaze were assigned 64GB of RAM, and Virtuoso was set up with 64GB or more as recommended. pmr was allowed 32GB buffer for handling the queries. Since we run large batches of queries (100+), these are executed in succession in order to simulate a realistic load to a database system.

Querying Wikidata. To gauge the potential of PMRs on real-world applications, we use WDBench [3], a recently proposed Wikidata query benchmark. WDBench uses a streamlined version of the Wikidata knowledge graph [60], and a curated set of real-world user-posted queries from the Wikidata endpoint public query log [41]. The WDBench Wikidata graph contains a total of 364.6M nodes and 1.257B edges. WDBench proposes multiple sets of real-world benchmark queries. One of these is a set of RPQs, containing 660 RPQs in total. From this set we select those that have at least one endpoint of the path defined, leaving us with 576 GRPQs. We run each query under the five different query modes (M1)–(M5) described earlier. Similarly to WDBench, we ran modes (M1), (M2), and (M3) with a limit of 100,000 results. For version (M3) this means that two different paths reaching the same endpoint count as two results. Modes (M4) and (M5) return all results. All queries were given a timeout of 1 minute (i.e., same timeout as the Wikidata SPARQL endpoint, and as specified by WDBench).

Results concerning version (M1) are presented in Table 2, where we see that pmr outperforms the competition *even when queries are only evaluated under the standard endpoint semantics.* This highlights the potential of PMRs not only for queries that return paths, but also for standard endpoint queries.

For the other evaluation modes (M2)–(M5) we note that among the four competitors, only Neo supports returning paths and shortest paths. As such, we only compare to Neo for these modes in what follows. For modes (M2)–(M4), only 315 queries could be expressed in Neo. Unfortunately, *all these queries timed out*, except two. Coincidentally, these two queries did not return any result because Neo detected that the start node given by the query is not in the database. The lower part of of Table 2 therefore shows results only for pmr, which exhibits a fairly stable behavior with very few timeouts. Both average and median times are reasonable given the magnitude of the dataset. We note that versions (M1)–(M3) have lower averages than (M4) and (M5). This is due to the 100k LIMIT applied to (M1)–(M3) which avoids constructing the full product graph, and which is not applied in (M4)–(M5).

The performance of Neo on modes (M2)–(M4) compared to pmr leads us to conclude that *(1) today's engines easily time out for queries that return paths on large-scale data* and *(2) PMRs have a significant potential to improve this situation.*

Measuring Scalability. Next, we investigate how pmr scales compared to Neo as a function of path lengths. To that end, we consider a controlled, synthetic set of queries that involve paths, parametrized by a start node s and number $k \in \mathbb{N}$.

- Return all nodes t that are reachable from s. **(Q1)**
- For each node t reachable from s by a path ρ of length $\leq k$,
  - return a single such shortest path ρ; **(Q2a)**
  - return all such shortest paths ρ; **(Q2b)**
  - count the number of such shortest paths ρ. **(Q2c)**
- For each node t reachable from s by a path ρ of length $= k$
  - return a single such path ρ; **(Q3a)**
  - return all such paths ρ; **(Q3b)**
  - count the number of such paths ρ. **(Q3c)**

We feel that these queries are fundamental to graph database systems. For instance, Q2a and Q3a give users an idea of why a node is reachable. Queries Q2b and Q3b may be typical subqueries in a larger query plan, where the user still wants to further investigate the paths, e.g., for graph analytics. (This is, after all, an important reason why Cypher and GQL allow such queries in the first place.) Queries Q2c and Q3c are perhaps the most fundamental analytical test one can do with the paths that connect nodes: counting the number of paths to t of a given length gives us a rough idea of how well s and t are connected. Furthermore, notice that we ask for shortest paths in the Q2 queries, whereas shortest paths don't play a role in the Q3 queries.

For queries Q1, Q2a, Q2b, Q3a and Q3b, we set a limit of 100,000 paths and we use the same 1 minute timeout as before. We evaluate these queries on the following two data sets:

DMND: the graph in Figure 2 with $n = 1000$; and
FB: the ego-Facebook data set from SNAP [38], containing 4,039 nodes and 176,468 edges[5].

On DMND, we chose s to be the leftmost node in Figure 2 and on FB we selected the node with id 0 in the dataset. These initial nodes are maintained for all the queries.

Notice that Q1 only returns *nodes*, i.e., no paths are returned. Still, we see in Table 3 that pmr drastically outperforms Neo: it is faster than 0.2s whereas Neo times out at one minute. The Cypher query we ran was essentially

```
MATCH ({id:"N0"})-[*]->(x) RETURN DISTINCT x LIMIT 100k
```

If we changed it, however, to **Q1'**, being

```
MATCH ({id:"N0"})-[*]->(x) RETURN x LIMIT 100k
```

[5]The original graph is undirected, which we modeled with edges in both directions.

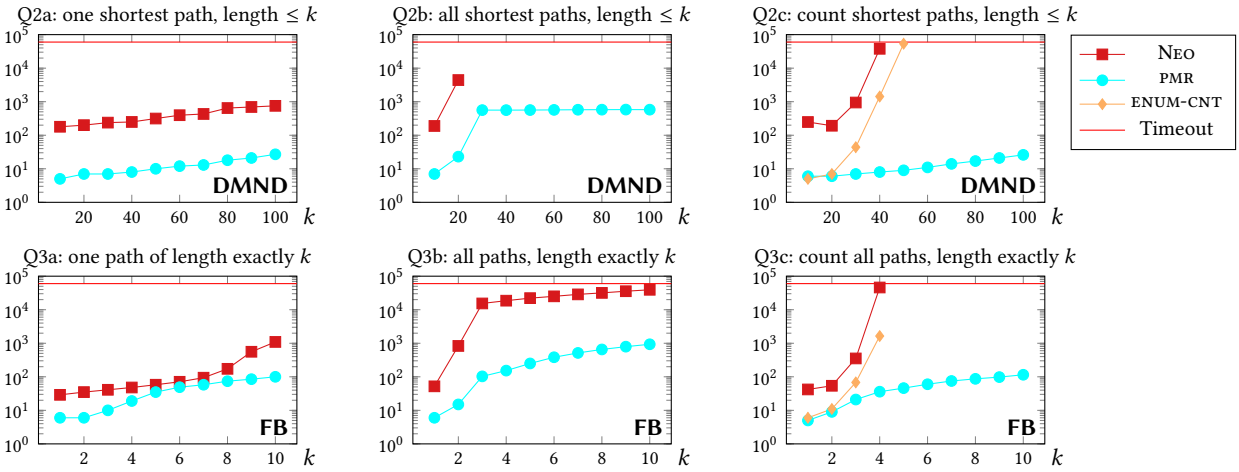


Figure 8: Run-time of Q2a–Q2c on the synthetic DMND data and of Q3a–Q3c on the real-world FB data (log scale, in ms).

the run-time in Neo improved, but was still 620× slower than PMR on DMND and 55× on FB. Per Cypher semantics, Q1' returns each node x as often as there is a trail from s to x. It runs faster in Neo than Q1 because the multiplicities cause it to reach the 100,000 limit faster than Q1. Notice that Table 3 also shows the performance of the SPARQL engines on Q1. Even though they do not build a data structure to return paths, they are outperformed by PMR.

Synthetic Data (DMND). Figure 8 (top row) summarizes the run-times of queries Q2a–Q2c on DMND. (Queries Q3a–Q3c, which give a similar picture, are omitted for space reasons; we show them for the next data set.) Concerning Q2a, PMR is about 30× faster than Neo. For Q2b, Neo already times out when finding all the shortest paths of length ≤ 30, whereas PMR stabilizes at ~0.5s due to our limit of 100,000 paths.

The most drastic differences are observed for counting, i.e., query Q2c. Whereas PMR's run time ranges from 6–26 ms, Neo is around 38,000 ms already when $k = 40$ and times out afterwards. To get some insight in the inherent complexity of the evaluation strategy of Neo, we implemented a different evaluation strategy in PMR: instead of counting paths using the PMR (Proposition 5.11), we simply enumerate paths one-by-one and count them. This is indicated as ENUM-CNT in Figure 8. Here, the difference between the exponential-time algorithm for ENUM-CNT and the polynomial-time approach of PMR is clearly visible, and ENUM-CNT scales similarly to Neo.

Social Netword Data (FB). Figure 8 (bottom row) shows the run-times of Q3a–Q3c on FB. (Queries Q2a–Q2c show similar behavior and are omitted for space reasons.) On Q3a, Neo and PMR perform similarly up to $k = 7$. From then on, Neo's performance degrades quickly, while PMR steadily increases from 58ms to 100ms. The reason for this difference is that the diameter of FB is 8: when we search for paths of length 7 and longer, we start considering paths with duplicate nodes. In terms of performance, it seems that PMR handles such situations better than Neo. For query Q3b, we see a similar behavior as in DMND. Both systems exhibit a fast increase in the beginning until the limit of 100,000 paths is reached. From then on, there is a more gradual increase due to the increasing complexity of finding paths of length *exactly* k.

Again, we see the most drastic difference in the counting query, i.e., when we use PMRs to represent the intermediate result of a the query, over which we then compute an aggregate value. Whereas Neo reaches 46,000 ms at $k = 4$ and times out afterwards, PMR is at 36 ms for $k = 4$ (i.e., improving by 3 orders of magnitude) and remains at 115 ms for $k = 10$.

We therefore feel that PMRs have the potential to *drastically improve the scalability of modern graph pattern matching queries.*

7 RELATED WORK

Queries over graph-structured data have been extensively studied, e.g. [1, 17, 20, 30, 45]. A popular means of querying are conjunctive regular path queries (CRPQs) [17, 25, 28, 30], which return tuples of nodes which are connected in a way predefined by the CRPQ. This mode of evaluating (conjunctive) regular path queries has dominated the research landscape for decades [10] and is also the mode of evaluation for regular path queries in SPARQL [6, 39, 63].

However, as the data gets larger and more complex, it gets more and more important to include paths in the output of the query [37]. Indeed, G-Core [4], a result of intense collaboration between industry and academia, proposes to treat paths first-class citizens in graph databases and, hence, allows queries to return them. GQL [24], the upcoming ISO standard for querying property graphs, builds on the G-Core proposal, but takes a perspective closer to industry. The industry/academia collaboration for G-Core and GQL takes place under the auspices of the LDBC, which also generated work on keys for property graphs [5] and threshold queries [13].

The three lines of work that are the most closely connected to ours are the following.

Factorized databases. Olteanu and co-authors have proposed Factorized Database Representations (FDBs) [8, 9, 48, 49] as a means of succinctly representing query results, possibly exponentially more succinct than traditional tables, while still allowing enumeration of such tables with constant delay. Factorized databases hence share important properties with the path multiset representation proposed here. We stress, however, that FDBs and PMRs are incomparable. Indeed, on the one hand PMRs are more expressive

than FDBs: FDBs were developed to represent results of traditional conjunctive queries on relational databases (or, more generally, relational algebra queries), not for representing results of GRPQs applied to graphs. In particular, conjunctive queries, when evaluated on graphs, can only return paths whose length is bounded by the number of atoms in the query. By contrast, GRPQs can return paths of unbounded length. Consequently FDBs can only represent paths of bounded length, while PMRs can represent paths of unbounded length.

On the other hand, FDBs are more expressive than PMRs. This is because FDBs can represent results of any conjunctive query, and, on graphs, conjunctive queries can express patterns such as triangles that do not adhere to a path topology. While FDBs can represent such expressive graph patterns, PMRs are limited to paths.

Finally, FDBs and PMRs are fundamentally distinct mathematical objects. FDBs represent relational tables as an expression involving unions and Cartesian products, whereas PMRs are graphs, endowed with a homomorphism.

Finite state automata and ECRPQs. Some of our constructions (notably Definition 5.1) are heavily inspired on the product construction for non-deterministic finite automata [33]. Indeed, taking the "product" of a graph and an NFA is a folklore method for computing the output of regular path queries in the literature. The literature, however, usually deals with *sets of endpoint pairs*, which is easier than multisets of paths. Barceló et al. [11] used a different but similar construction to investigate query evaluation for *extended conjunctive regular path queries (ECRPQs)* which, as us, also extend CRPQs with the ability to include paths in the output of the query, but also to define complex semantic relationships between paths, using regular relations. Like us, they provide an automaton construction that can represent both nodes and paths in the output. The remainder of their work is quite different from ours, since they had a different focus. They provided a picture of what can be implemented in standard query languages in terms of complexity, including concerning questions such as query containment. To deal with relations on paths, they define a notion of convolutions of graph databases and queries, that reduces the evaluation of ECRPQs to the evaluation of CRPQs. Our focus on compact representations, and their interaction with modular operations in query plans, is therefore quite different.

Graph compression. In query-preserving graph compression (e.g., [19, 27]) as well as in work on structural indexing for graphs (e.g. [18, 36]) the goal is to compress input graph G into a smaller graph C such that for every query Q in a fixed class of queries $\mathcal{Q}$ we have $Q(G) = Q(C)$, i.e., Q gives the same answer on C as on the original graph G. As such, one can use C instead of G to answer queries in $\mathcal{Q}$ on G, which is more efficient. Graph compression and structural indexing is orthogonal to our work for two reasons. (1) We aim to compactly represent the *output* $Q(G)$ of a single RPQ Q on G, while in [18, 27, 36] the aim is to compress the *input* graph G w.r.t a class of queries $\mathcal{Q}$. (2) We consider queries that return paths while [18, 27, 36] consider endpoint queries (in the form of reachability queries) or bisimulation matchings. This difference is important, since compression then needs to preserve endpoints or matchings, but not the paths themselves. In particular, the techniques in [18, 27, 36] *contract* paths under bounded bisimulation during compression. This does not preserve paths, and as such $Q(G) = Q(C)$ does not necessarily hold when Q is a GRPQ that returns paths.

8 CONCLUSIONS

We presented the concept of *path multiset representations (PMRs)*, which allow to represent multisets of paths in asuccinct manner. We believe that such a concept is necessary for ensuring that returning paths in graph query engines remains tractable. Indeed, our experiments show that today's engines are not ready to deal with queries that return paths, and that PMRs can improve run-times by orders of magnitude. Theoretically we prove that, while the number of paths or shortest paths that match regular path queries can become prohibitively large, PMRs allow to represent these using linear space in terms of combined complexity.

This paper presents a wide number of results that involve the incorporation of PMRs in graph engines, using a modular query evaluation approach typical of how database systems work. By showing how PMRs hold up when considering grouping operators, unions, projection, counting, and random sampling, we have gone significantly beyond the restricted setting that is typically considered in research, i.e., regular path queries and set semantics.

PMRs may even be useful in terms of query language design. An important reason why selectors and restrictors to finite sets of paths are used in in modern graph query languages [23, 24] is because the community does not know how to deal with infinite sets of paths. But such restrictions can be detrimental to query languages. For instance, by restricting ourselves to data structures that can only represent finite sets of paths, we intuitively make logical and physical operators less composable, which in turn may rule out operations further in the query plan. For example, it is not possible to randomly sample a path of length n between two nodes, if we have discarded the paths of this length in a previous computation step. PMRs, however, can represent the infinite sets that are returned by regular path queries in a finite manner, as Example 3.3 and Theorem 5.2 illustrate. It is therefore an interesting question whether a composable algebra for graph querying that allows infinite intermediate results can be built up using PMRs or a variation thereof.

ACKNOWLEDGMENTS

We are grateful to Matthias Hofer for valuable discussions and to Wojciech Czerwiński for pointing us to [58]. This work was supported by the ANR project EQUUS ANR-19-CE48-0019; funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 431183758. Vansummeren was supported by the Bijzonder Onderzoeksfonds (BOF) of Hasselt University (Belgium) under Grant No. BOF20ZAP02. Vrgoč and Rojas were supported by ANID – Millennium Science Initiative Program – Code ICN17_002. Vrgoč was also supported by ANID Fondecyt Regular grant nr. 1221799.

REFERENCES

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. 1997. The Lorel Query Language for Semistructured Data. *Int. J. Digit. Libr.* 1, 1 (1997), 68–88.

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

[3] Renzo Angles, Carlos Buil Aranda, Aidan Hogan, Carlos Rojas, and Domagoj Vrgoč. 2022. WDBench: A Wikidata Graph Query Benchmark. In *The Semantic Web - ISWC 2022 - 21st International Semantic Web Conference, Virtual Event, October 23-27, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13489)*, Ulrike Sattler, Aidan Hogan, C. Maria Keet, Valentina Presutti, João Paulo A. Almeida, Hideaki Takeda, Pierre Monnin, Giuseppe Pirrò, and Claudia d'Amato (Eds.). Springer, 714–731. https://doi.org/10.1007/978-3-031-19433-7_41

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

[5] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *International Conference on Management of Data (SIGMOD)*. ACM, 2423–2436.

[6] 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 *International Conference on World Wide Web (WWW)*. 629–638.

[7] Guillaume Bagan, Angela Bonifati, and Benoît Groz. 2013. A Trichotomy for Regular Simple Path Queries on Graphs. In *Symposium on Principles of Database Systems (PODS)*. 261–272.

[8] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. 2013. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.* 6, 14 (2013), 1990–2001. https://doi.org/10.14778/2556549.2556579

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

[10] Pablo Barceló. 2013. Querying graph databases. In *Symposium on Principles of Database Systems (PODS)*. 175–188.

[11] Pablo Barceló, Leonid Libkin, Anthony Widjaja Lin, and Peter T. Wood. 2012. Expressive Languages for Path Queries over Graph-Structured Data. *ACM Transactions on Database Systems* 37, 4 (2012), 31:1–31:46.

[12] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. 2020. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News* 7, 1 (2020), 4–33.

[13] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Slawek Staworko, and Dominik Tomaszuk. 2022. Threshold Queries in Theory and in the Wild. *Proc. VLDB Endow.* 15, 5 (2022), 1105–1118.

[14] Angela Bonifati, Wim Martens, and Thomas Tim. 2019. Navigating the Maze of Wikidata Query Logs. In *The Web Conference (WWW)*. ACM. To appear.

[15] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679.

[16] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 1999. Rewriting of Regular Expressions and Regular Path Queries. In *ACM Symposium on Principles of Database Systems*. ACM Press, 194–204.

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

[18] Qun Chen, Andrew Lim, and Kian Win Ong. 2003. D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 134–144. https://doi.org/10.1145/872757.872776

[19] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On Compressing Social Networks. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Paris, France) *(KDD '09)*. Association for Computing Machinery, New York, NY, USA, 219–228. https://doi.org/10.1145/1557019.1557049

[20] Mariano P. Consens and Alberto O. Mendelzon. 1990. GraphLog: a Visual Formalism for Real Life Recursion. In *Symposium on Principles of Database Systems (PODS)*. 404–416.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company.

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

[23] cypher [n.d.]. Cypher Query Language. https://neo4j.com/developer/cypher/.

[24] 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 Vrgoč, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD '22*. 2246–2258. https://doi.org/10.1145/3514221.3526057

[25] Alin Deutsch and Val Tannen. 2001. Optimization Properties for Classes of Conjunctive Regular Path Queries. In *International Workshop on Database Programming Languages DBPL (Lecture Notes in Computer Science, Vol. 2397)*. Springer, 21–39.

[26] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35, 1 (2012), 3–8. http://sites.computer.org/debull/A12mar/vicol.pdf

[27] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query preserving graph compression. In *SIGMOD Conference*. ACM, 157–168.

[28] Diego Figueira, Adwait Godbole, Shankara Narayanan Krishna, Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. Containment of Simple Conjunctive Regular Path Queries. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 371–380.

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

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

[31] 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 *SIGMOD Conference*. ACM, 1433–1445.

[32] GSQL [n.d.]. GSQL. https://www.tigergraph.com/gsql/.

[33] J.E. Hopcroft, R. Motwani, and J.D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation* (3 ed.). Addison-Wesley.

[34] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *International Conference on Management of Data (SIGMOD)*. ACM, 1259–1274.

[35] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *VLDB J.* 29, 2-3 (2020), 619–653.

[36] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. 2002. Exploiting local similarity for indexing paths in graph-structured data. In *Proceedings 18th International Conference on Data Engineering*. 129–140. https://doi.org/10.1109/ICDE.2002.994703

[37] Krys J. Kochut and Maciej Janik. 2007. SPARQLeR: Extended Sparql for Semantic Association Discovery. In *ESWC (Lecture Notes in Computer Science, Vol. 4519)*. Springer, 145–159.

[38] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford large network dataset collection.

[39] Katja Losemann and Wim Martens. 2013. The complexity of regular expressions and property paths in SPARQL. *ACM Transactions on Database Systems* 38, 4 (2013), 24:1–24:39.

[40] Katja Losemann and Wim Martens. 2014. MSO queries on trees: enumerating answers under updates. In *Joint Meeting of the Conference on Computer Science Logic (CSL) and the ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 67:1–67:10.

[41] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International Semantic Web Conference (ISWC)*. 376–394.

[42] Wim Martens, Matthias Niewerth, Tina Popp, Stijn Vansummeren, and Domagoj Vrgoč. 2022. Representing Paths in Graph Database Pattern Matching. *CoRR* abs/2207.13541 (2022). https://arxiv.org/abs/2207.13541

[43] Wim Martens, Matthias Niewerth, and Tina Trautner. 2020. A Trichotomy for Regular Trail Queries. In *STACS (LIPIcs, Vol. 154)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:16.

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

[45] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (12 1995), 1235–1258.

[46] Neo4j [n.d.]. Neo4j. neo4j.com.

[47] Neo4j. 2019. The Neo4j Developer Manual v3.4. https://neo4j.com/docs/developer-manual/3.4/.

[48] Dan Olteanu. 2020. The Relational Data Borg is Learning. *Proc. VLDB Endow.* 13, 12 (2020), 3502–3515. https://doi.org/10.14778/3415478.3415572

[49] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1 (2015), 2:1–2:44. https://doi.org/10.1145/2656335

[50] PGQL [n.d.]. PGQL. https://pgql-lang.org/.

[51] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz,

Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. 2021. The future is big graphs: a community view on graph processing systems. *Commun. ACM* 64, 9 (2021), 62–71.

[52] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. 2022. Enumeration for FO Queries over Nowhere Dense Graphs. *J. ACM* 69, 3 (2022), 22:1–22:37. https://doi.org/10.1145/3517035

[53] Luc Segoufin. 2013. Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, Wang-Chiew Tan, Giovanna Guerrini, Barbara Catania, and Anastasios Gounaris (Eds.). ACM, 10–20. https://doi.org/10.1145/2448496.2448498

[54] Jena Team. 2022. TDB Documentation. https://jena.apache.org/documentation/tdb/

[55] Stardog Team. 2021. Stardog 7.6.3 Documentation. https://docs.stardog.com/

[56] Bryan B. Thompson, Mike Personick, and Martyn Cutcher. 2014. The Bigdata® RDF Graph Database. In *Linked Data Management*, Andreas Harth, Katja Hose, and Ralf Schenkel (Eds.). Chapman and Hall/CRC, 193–237. http://www.crcnetbase.com/doi/abs/10.1201/b16859-12

[57] TigerGraph [n.d.]. TigerGraph. www.tigergraph.com.

[58] Wen-Guey Tzeng. 1996. On Path Equivalence of Nondeterministic Finite Automata. *Inf. Process. Lett.* 58, 1 (1996), 43–46.

[59] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *Proc. VLDB Endow.* 13, 9 (2020), 1582–1597.

[60] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85. https://doi.org/10.1145/2629489

[61] Domagoj Vrgoč. 2022. Evaluating regular path queries under the all-shortest paths semantics. *CoRR* abs/2204.11137 (2022). https://doi.org/10.48550/arXiv.2204.11137

[62] Domagoj Vrgoč, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2021. MillenniumDB: A Persistent, Open-Source, Graph Database. *CoRR* abs/2111.01540 (2021). https://arxiv.org/abs/2111.01540

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