# Path-based Algebraic Foundations of Graph Query Languages

Renzo Angles
Universidad de Talca & IMFD Chile
Chile
renzoangles@gmail.com

Angela Bonifati
Lyon 1 Univ., Liris CNRS & IUF
France
angela.bonifati@univ-lyon1.fr

Roberto García
Universidad de Talca & IMFD Chile
Chile
roberto.garcia@utalca.cl

Domagoj Vrgoč
PUC Chile & IMFD Chile
Chile
vrdomagoj@uc.cl

## ABSTRACT

Graph databases are gaining momentum thanks to the flexibility and expressiveness of their data models and query languages. A standardization activity driven by the ISO/IEC standardization body is also ongoing and has already conducted to the specification of the first versions of two standard graph query languages, namely SQL/PGQ and GQL, respectively in 2023 and 2024. Apart from the standards, there exists a panoply of concrete graph query languages provided by current graph database systems, each offering different query features. A common limitation of current graph query engines is the absence of an algebraic approach for evaluating path queries. To address this, we introduce an abstract algebra for evaluating path queries, allowing paths to be treated as first-class entities within the query processing pipeline. We demonstrate that our algebra can express a core fragment of path queries defined in GQL and SQL/PGQ, thereby serving as a formal framework for studying both standards and supporting their implementation in current graph database systems. We also show that evaluation trees for path algebra expressions can function as logical plans for evaluating path queries and enable the application of query optimization techniques. Our algebraic framework has the potential to act as a lingua franca for path query evaluation, enabling different implementations to be expressed and compared.

## 1 INTRODUCTION

Graph databases are becoming a widely spread technology, leveraging the property graph data model, and exhibiting great expressiveness and computational power [4]. The success of graph data systems such as Neo4j, TigerGraph, MemGraph, Oracle PGX, AWS Neptune and RedisGraph had led to a standardization activity around graph query languages, carried out by the ISO/IEC standardization body. The ISO/IEC has already finalized the first version of SQL/PGQ [17] as part of the 2023 version of the SQL standard and has recently finalized GQL [18], a native graph query language that will eventually not only return tables but also paths and graphs.

Finding and returning paths is a fundamental part of every graph query language as witnessed by the rich set of features for manipulating paths in the SQL/PGQ and GQL standard. However, while the ISO standards do prescribe mechanisms for path manipulation, current engines are severely lagging in their implementation. Indeed, to the best of our knowledge, there is currently no engine that fully supports the path features prescribed by the two standards, and most solutions deploy their own definitions to express and evaluate path queries [8]. We believe that one reason for this is the lack of a common algebra that allows one to express the multitude of path query features required by graph engines and prescribed by the SQL/PGQ and GQL standards.

Therefore, in this paper, we lay the foundations of a path-based algebraic framework for evaluating path queries. Our effort is relevant from both a theoretical viewpoint and a system perspective. In fact, a standard graph query algebra is missing, while it is a core component of next-generation graph ecosystems and their user cases [30]. Our framework is *expressive* as it encompasses the fundamental path features of current graph query languages and the ISO standards while precisely formalizing their semantics. It also offers *query composability*, allowing one to specify algebraic expressions that can be arbitrarily nested and returns sets of paths that are consumed by other queries. It also exhibits *strict adherence* to the standard graph query languages in terms of the covered query operators and the different variants of the query semantics supported by them. It also embodies a blueprint for the *algebra-based implementation* of graph queries across systems, since it directly compiles into logical plans to evaluate graph queries, paving the way to the final adoption of the graph query language standards themselves. Additionally, our formalization of the path-based algebra *anticipates* the future versions of the query language standards, expressing several properties outside of their current scope.

To illustrate the expressiveness of our path-based algebraic framework, we introduce an example below.

*Path algebra by example.* Consider the property graph shown in Figure 1, which is a snippet of the graph provided by the LDBC Social Network Benchmark [31], a popular benchmark for graph database systems. The graph contains two types of nodes (identified by the labels `Person` and `Message`) and three types of edges (identified by the labels `Knows`, `Likes` and `Has_creator`). An essential characteristic of this graph is the capability to employ recursion, due to the presence of cycles. Specifically, there is an inner cycle that involves edges `Knows` and an outer cycle that traverses the concatenation of edges `Likes` and `Has_creator`.

An example of a path query (in GQL-like syntax) leveraging the cyclic structure of the underlying graph data is reported below. The query computes all the paths from the node `n1` (`Moe`) to the node `n4` (`Apu`), either across the inner cycle with label `Knows` or across the outer cycle with labels `Likes` and `Has_creator`.

```
MATCH  p = (x {name:"Moe"})-[(Knows+)|
           (Likes/Has_creator)+]->(y {name:"Apu"})
```

In this paper, we introduce a comprehensive path algebra that allows us to evaluate path queries as relational database systems

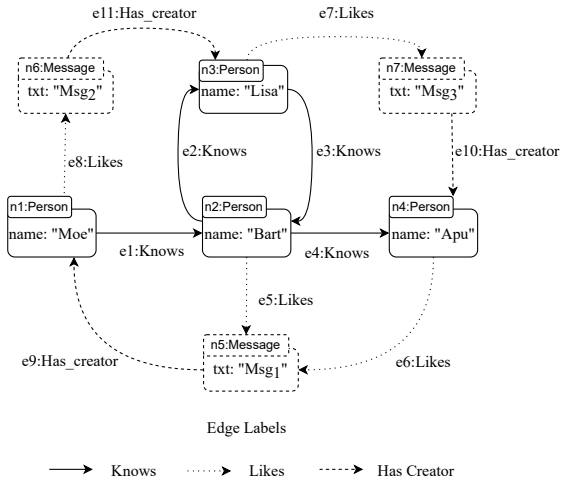



**Figure 1: A graph representing a social network (drawn from the LDBC SNB benchmark).**

do for SQL queries. Specifically, given a path query expression, we are able to create a logical query plan, and then a physical query plan. For example, Figure 2 shows an evaluation tree that represents a logical query plan for the query presented above.

In general, our algebra mimics the standard relational algebra in terms of symbols used, but it operates on sets of paths instead of relations. As such, it uses the set of nodes and the set of edges (i.e. paths of length zero and one, respectively) as its atoms, and combines them to construct and filter out paths. Our algebra is divided into three groups: core path algebra, recursive path algebra, and extended path algebra.

The *core algebra* includes three operators: selection, join and union. The *selection* operator ($\sigma$) filters a set of paths according to a specific selection condition. For example, $label(edge(1))$ = `Likes` filters the paths whose first edge has the label `Likes`. The *union* operator ($\cup$) computes the union of two sets of paths, eliminating duplicates. The *join* operator ($\bowtie$) combines paths from two sets by generating a new path for each pair of paths $p_1$,$p_2$ satisfying that the final node of $p_1$ is equal to the first node of $p_2$, thus mimicking the concatenation of paths.

The *recursive path algebra* is based on the recursive operator $\phi$, which computes a recursive self-join over a set of paths, allowing the construction of paths of arbitrary length. Returning to the example shown in Figure 2, the underlying semantics of the $\phi_{Walk}$ operator is to construct paths by applying a recursive concatenation of paths, starting with paths of the form ($a$, `Knows`, $b$), without any type of restriction on the resulting paths. In this case, due to the presence of the two cycles in the graph shown in Figure 1 (the inner loop formed by the label `Knows`, and the outer loop formed by the concatenation of the label `Likes` with the label `Has_creator`), the query will never halt, since it can keep on looping and returning longer and longer paths.

To cope with the issue of infinite results, GQL and SQL/PGQ impose a tight policy on paths that can be returned through the concept of *restrictors*, that control the type of paths that are matched to the query (for instance, shortest walks or simple paths). Our algebra mimics this behavior by specializing the $\phi$ operator according to different semantic restrictions that must be imposed. Specifically, in addition to the Walk semantics ($\phi_{Walk}$), our algebra provides recursive operators for the Acyclic, Simple, Trail, and Shortest path semantics. All these semantics are included in the core pattern matching fragment of both GQL and SQL/PGQ, which is common to the two standards. Hence, if we change the recursive operators in our example query tree with $\phi_{Simple}$, then the result of the query will only contain the following two paths:

$$path_1 = (n_1, e_1, n_2, e_4, n_4)$$
$$path_2 = (n_1, e_8, n_6, e_{11}, n_3, e_7, n_7, e_{10}, n_4),$$

where we denote a path as an interchanging sequence of nodes and edges, starting and ending with a node.

The *extended path algebra* introduces the notion of a solution space and defines three operators (similar to those in SQL but designed for path manipulation): group-by, order-by and projection. A *solution space* is a data structure used to organize a set of paths into groups which are further organized into partitions.

The *group-by* operator receives a set of paths and generates a solution space where the paths, groups, and partitions can be organized in 8 different ways (e.g. multiple partitions, with a single group per partition, such that all the paths in a group have the same initial node). The *order-by* operator sorts the paths, groups, and partitions of a solution space based on the length of the paths. The *projection* operator returns a set of paths extracted from a solution space according to a given criterion. The components of the extended algebra were designed to support different types of *Selectors* (e.g. `ANY SHORTEST`), which is a novel feature introduced in GQL and SQL/PGQ.

A key feature of our algebra is query composability, as a set of paths serves as the primary data structure for input and output in the algebra operators (with solution spaces as secondary data structures). It is important to note that current graph query languages are unable to manipulate a set of paths, and query composability is often lost when returning paths as bindings.

Overall, our contributions can be summarized as follows:

- We introduce an abstract algebra for evaluating regular path queries, allowing paths to be treated as first-class entities within the query processing pipeline.
- We demonstrate that our algebra can express a core fragment of path queries defined in GQL and SQL/PGQ, and can therefore serve as a formal framework for studying both standards.
- We provide precise and concrete semantics for the selectors and restrictors introduced in GQL and SQL/PGQ. Additionally, we include several operators missing from the two proposals (e.g. projection), providing space for future additions to the standard.
- We also show that evaluation trees for path algebra expressions can function as logical plans for evaluating path queries and enable the application of query optimization techniques. Concretely, once we have an algorithm for each operator in the algebra, a sound proof of concept implementation of the GQL and SQL/PGQ standards can be provided with ease.
- We provide an open-source parser of the algebra and we make it publicly available for the wider community.
- Our algebraic framework has the potential to act as a lingua franca for path query evaluation, enabling different implementations to be expressed and compared.

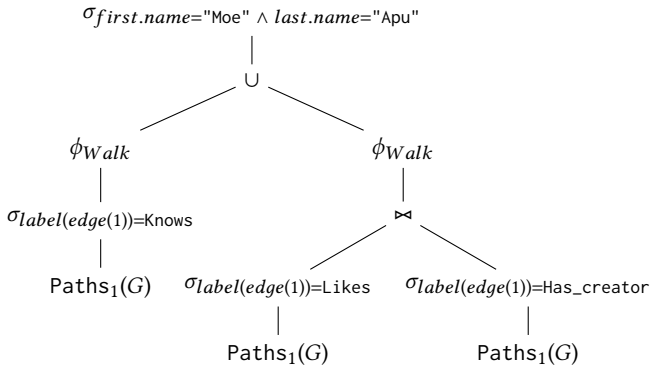



**Figure 2: Example of a query tree based on a path algebra expression that allows the evaluation of a recursive path query.**

## 2 PRELIMINARIES

The path algebra proposed in this article has been designed to return and manipulate sets of paths. In this sense, each algebra operator takes one or two sets of paths as input, and its evaluation returns a single set of paths. Next, we introduce basic concepts associated to property graphs and paths.

### 2.1 Property graphs

Informally, a property graph is a directed labeled multigraph with the special characteristic that each node or edge could maintain a (possibly empty) set of property-value pairs [1]. From a data modeling point of view, a node represents an entity, an edge represents a relationship between entities, and a property represents a specific feature of an entity or relationship.

Formally, let **O** be an infinite set of object identifiers, **L** be an infinite set of labels, and **V** be an infinite set of values. Labels will be represented as unquoted strings without whitespace. Values will be represented as quoted strings.

*Definition 2.1 (Property graph).* A property graph is defined as a tuple G = $(N, E, \rho, \lambda, \nu)$ where:

(1) $N \subset \mathbf{O}$ is a finite set of node identifiers;
(2) $E \subset \mathbf{O}$ is a finite set of edge identifiers where $N \cap E = \emptyset$;
(3) $\rho : E \rightarrow (N \times N)$ is a total function that defines the pairs of nodes connected by each edge;
(4) $\lambda : (N \cup E) \rightharpoonup \mathbf{L}$ is a partial function that assigns labels to nodes and edges;
(5) $\nu : (N \cup E) \times \mathbf{L} \rightharpoonup \mathbf{V}$ is a partial function that defines properties for nodes and edges.

As an illustration, consider the graph shown in Figure 2. According to the above definition, we will have that $N = \{n_1, \ldots, n_7\}$ is the set of node identifiers, and $E = \{e_1, \ldots, e_{11}\}$ is the set of edge identifiers. Nodes and edges are considered objects in a graph. We see that $\lambda(n_1) =$ `Person` defines the node label of $n_1$, and $\lambda(e_1) =$ `Knows` defines the edge label of $e_1$. Given an object $o$ (node or edge), if $o \notin$ `Domain`$(\lambda)$ then $\lambda(o) = \emptyset$. Given the edge $e_1$ with $\rho(e_1) = (n_1, n_2)$, we say that $n_1$ and $n_2$ are the source node and the target node of $e_1$, respectively. The assignment $\nu(n_1,$ `name`$) =$ `"Moe"` indicates that the node $n_1$ has a property with the label `name` whose value is `"Moe"`.

### 2.2 Paths

A path $p$ in a property graph $G = (N, E, \rho, \lambda, \nu)$ is a sequence of node and edge identifiers of the form

$$p = (n_1, e_1, n_2, e_2, n_3, e_3, \ldots, e_k, n_{k+1})$$

where $k \geq 0$, $n_i \in N$, $e_i \in E$, and $\rho(e_i) = (n_i, n_{i+1})$ for $1 \leq i \leq k+1$. The last condition ensures that for each pair of edges $e_i$, $e_j$ in $p$, the target node of $e_i$ is equal to the source node of $e_j$.

The length of a path $p$ is the number of edge identifiers in $p$. The label of a path $p$, denoted as $\lambda(p)$, is the sequence of edge labels obtained from $p$, that is, $\lambda(p) = (\lambda(e_1), \ldots, \lambda(e_k))$.[1] If the length of $p$ is 0 then $\lambda(p) = ()$, i.e., an empty sequence of labels.

Given a graph $G = (N, E, \rho, \lambda)$, the function $Paths_0(G)$ returns the paths of length 0 in $G$, that is, $Paths_0(G) = \{(n) \mid n \in N\}$. The function $Paths_1(G)$ returns the paths of length 1, that is, $Paths_1(G) = \{(n, e, n') \mid e \in E \wedge \rho(e) = (n, n')\}$. The function $Paths_*(G)$ returns all the paths in $G$.[2]

Two paths are equal if they have the same sequence of node and edge identifiers. A path $p = (n_1, e_1, \ldots, e_k, n_{k+1})$ is called *acyclic*, if $n_i \neq n_j$, for all $i \neq j$, and it is called *simple* if $n_i \neq n_j$ for all $i \neq j$, except that we allow $n_1 = n_{k+1}$, which means that the start and end nodes can be the same. A path is a *trail*, if $e_i \neq e_j$, for all $i \neq j$. Finally, we remark that, following the theoretical graph literature, GQL and SQL/PGQ use the term *walk* to indicate an arbitrary path.

### 2.3 SQL/PGQ and GQL

In this section, we provide a concise recap of the formalization of GQL [18] and SQL/PGQ [17] path queries. Path queries in both standards are based on *path patterns*, which are an extension of regular path queries (RPQs) [7], which are expressions of the form $(x, r, y)$, where $x, y$ are variables or constants, and $r$ is a regular expression. This query then returns all pairs of nodes $(n, n')$ in a property graph that are linked by a path whose edge labels form a word that matches the regular expression $r$.

Although in the research literature, RPQs only look for nodes and not for paths [2], in GQL and SQL/PGQ, it is also important to retrieve paths witnessing these connections. Of course, as illustrated in the introductory example, in the presence of cycles there is a potentially infinite number of such paths. To cope with this issue, GQL and SQL/PGQ introduce *selectors* and *restrictors* as a way to select the paths to be returned and to specify the semantics used for computing the paths, respectively. For example, consider the following path query in GQL,

```
ANY SHORTEST WALK p = (x)-[Knows]->+(y),
```

where `ANY SHORTEST` is the selector clause and `WALK` is the restrictor clause. In this case, the restrictor indicates that the query

[1] We use this definition because a path query is based on exploring edge labels.
[2] Breadth First Search (BFS) and Depth First Search (DFS) are two well-know algorithms that can be used to obtain all the paths in a graph. However, they need to assume specific "path semantics" to ensure finiteness.

will compute the paths between any pair of people, connected by edges labeled Knows, one or more times, without any kind of restriction (i.e. arbitrary path semantics). Additionally, the selector indicates that among all the retrieved paths, the query must return just a single shortest path, selected randomly. The allowed selectors and restrictors, and their corresponding semantics are presented in Table 1 and Table 2 respectively.

Following [8, 10], we can define a *path query* in GQL and SQL/PGQ as an expression of the form

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

In general terms, a query will return all the paths that match the specified selector-restrictor combination while at the same time being an answer to the underlying path pattern. We remark that the selector part is optional, with the restriction that for the WALK restrictor the selector must be specified in order to ensure a finite answer set.

In addition to plain path queries, GQL and SQL/PGQ allow concatenating two path queries into a sequence. For instance we can write $s\ r[s_1\ r_1\ (x, regex_1, y)] \cdot [s_2\ r_2\ (z, regex_2, w)]$, where $s, s_1, s_2$ are selectors, $r, r_1, r_2$ restrictors, and the query basically concatenates (when possible) paths in the answer of $s_1\ r_1\ (x, regex_1, y)$ and $s_2\ r_2\ (z, regex_2, w)$ and applies the $s\ r$ selector-restrictor combination to that set. This in particular means that we can ask for all trails connecting nodes $n_1$ and $n_2$, then all shortest walks connecting $n_2$ to $n_3$, and require that the entire concatenated path between $n_1$ and $n_3$ be a shortest trail. Another option allowed by GQL is taking the union of such answer sets, with the usual set-union semantics.

Finally, we remark that some functionalities of GQL such as group variables [10] are not covered in this paper, but given that these are used to collect nodes or edges along a path into a list, incorporating them into our framework is rather straightforward.

# 3 CORE PATH ALGEBRA

Given the sample graph shown in Figure 1, suppose that we would like to obtain the paths containing the friends and friends-of-friends of "Moe", i.e. the 1-hop and 2-hop paths. This question can be answered using the following GQL-like query:

```
MATCH p = (x {name:"Moe"})-[Knows|(Knows/Knows)]->(y).
```

In the above expression: (x {name:"Moe"}) denotes the source node, Knows|(Knows/Knows) is a regular expression, (y) denotes the target node, and p is a variable used to contain the resulting paths. The above declarative query can be transformed into an algebra expression whose evaluation tree is shown in Figure 3. Next, we explain the operators that conform the core of the path algebra proposed in this article. These are: selection, join and union.

Given a set of paths $S$, the *selection* operator ($\sigma$) allows us to filter the paths in $S$ according to a filter condition. In our example, the algebra expression $\sigma_{label(edge(1))=\texttt{Knows}}$ filters the paths in $\texttt{Paths}_1(G)$ (i.e. the paths of length one in $G$) such that the first edge of each path has the label Knows.

Given two sets of paths $\mathbb{S}_1$ and $\mathbb{S}_2$, the *join* operator ($\bowtie$) returns a set of new paths where each new path is the result of concatenating a path $p_1$ from $\mathbb{S}_1$ and a path $p_2$ from $\mathbb{S}_2$ such that the last node of $p_1$ is equal to the first node of $p_2$. In our example, the join operator for paths is used to obtain the paths with the structure $(n_1, \texttt{Knows}, n_2, \texttt{Knows}, n_3)$.

Following the usual semantics of the set theory, the *union* operator ($\cup$) combines two sets of paths into a single set of paths that includes all paths from the input sets. In our example, we use the union operator to combine the paths of the form $(n_1, \texttt{Knows}, n_2)$ with the paths of the form $(n_1, \texttt{Knows}, n_2, \texttt{Knows}, n_3)$.

Finally, the selection expression $\sigma_{first.name=\texttt{"Moe"}}$ in the root node of the evaluation tree allows filtering the paths returned by the union operator, such that each final path satisfies that its first node has a property name with value "Moe".

Note that the core algebra is closed under sets of paths, meaning that the input and the output of every operator is always a set of paths. It is very important because it allows compositionality and ensures that the output of every algebra expression is a set of paths. Next, we provide a formal definition of this core algebra.

## 3.1 Core Algebra - Formal definition

Given a path $p = (n_1, e_1, n_2, e_2, \ldots, e_k, n_{k+1})$, we define the following *path operators*:

- $\texttt{First}(p)$: returns the identifier of the first node occurring in $p$, e.g. $\texttt{First}(p) = n_1$;
- $\texttt{Last}(p)$: returns the identifier of the last node occurring in $p$, e.g. $\texttt{Last}(p) = n_{k+1}$;
- $\texttt{Node}(p, i)$: returns the identifier of the node occurring in the position $i$ of the path $p$, e.g $\texttt{Node}(p, 2) = n_2$;
- $\texttt{Edge}(p, j)$: returns the identifier of the edge occurring in the position $j$ of the path $p$, e.g $\texttt{Edge}(p, 1) = e_1$;
- $\texttt{Len}(p)$: returns the length (number of edges) of the path $p$, e.g. $\texttt{Len}(p) = k$;
- $\texttt{Label}(o)$: returns the label of an object (node or edge) $o$ occurring in $p$, e.g. $\texttt{Label}(\texttt{First}(p)) = \texttt{Person}$.
- $\texttt{Prop}(o, pr)$: returns the value of a property $pr$ of an object $o$, e.g. $\texttt{Prop}(\texttt{First}(p), name) = \texttt{"Lisa"}$.

Let $i \geq 1$ be an integer, $p$ be a path, $o \in O$ be an object, $v$ be a value, and $pr$ be a property name. A *selection condition* is defined recursively as follows. A simple selection condition is any of the expressions[3] $label(node(i)) = v$, $label(edge(i)) = v$, $label(first) = v$, $label(last) = v$, $node(i).pr = v$, $edge(i).pr = v$, $first.pr = v$, $last.pr = v$ and $len() = i$. If $c_1$ and $c_2$ are selection conditions, then $(c_1 \wedge c_2)$, $(c_1 \vee c_2)$, and $\neg(c_1)$ are complex selection conditions.

The evaluation of a selection condition $c$ on a path $p$, denoted $ev(c, p)$, returns True or False. A simple condition $c$ is evaluated as True in the following cases:

- if $c$ is $label(node(i)) = v$ and $\texttt{Label}(\texttt{Node}(p, i))$ returns $v$;
- if $c$ is $label(edge(i)) = v$ and $\texttt{Label}(\texttt{Edge}(p, i))$ returns $v$;
- if $c$ is $label(first) = v$ and $\texttt{Label}(\texttt{Node}(p, 1))$ returns $v$;
- if $c$ is $label(last) = v$ and $\texttt{Label}(\texttt{Node}(\texttt{Len}(p) + 1))$ returns $v$;
- if $c$ is $node(i).pr = v$ and $\texttt{Prop}(\texttt{Node}(p, i), pr)$ returns $v$;
- if $c$ is $edge(i).pr = v$ and $\texttt{Prop}(\texttt{Edge}(p, i), pr)$ returns $v$;
- if $c$ is $first.pr = v$ and $\texttt{Prop}(\texttt{Node}(p, 1), pr)$ returns $v$;
- if $c$ is $last.pr = v$ and $\texttt{Prop}(\texttt{Node}(p, \texttt{Len}(p) + 1), pr)$ returns $v$;
- if $c$ is $len() = i$ and $\texttt{Len}(p)$ returns $v$.

The evaluation of a complex selection condition is defined by following the usual semantics of propositional logic.

Let $p_1$ and $p_2$ be two paths satisfying $\texttt{Last}(p_1) = \texttt{First}(p_2)$. The *path concatenation* of $p_1$ and $p_2$, denoted $p_1 \circ p_2$, returns a new path composed by the sequence of $p_1$ followed by the tail of $p_2$ (i.e. the sequence of $p_2$ without the first node). For

[3] Our definition of simple selection conditions can be easily extended to support inequalities ($\neq, <, >, \leq$ and $\geq$) and other build-in functions (e.g., *substr* or *bound*).

| Expression | Informal semantics |
|---|---|
| `ALL` | Returns all paths, for every group, for every partition. |
| `ANY SHORTEST` | Returns one path with shortest length from each partition. **Non-Deterministic**. |
| `ALL SHORTEST` | Returns all paths in each partition that have the minimal length in the partition. **Deterministic**. |
| `ANY` | Returns one path in each partition arbitrarily. **Non-Deterministic**. |
| `ANY` $k$ | Returns arbitrary $k$ paths in each partition (if fewer than $k$ , then all are retained). **Non-Feterministic**. |
| `SHORTEST` $k$ | Returns the shortest $k$ paths (if fewer than $k$, then all are retained). **Non-Deterministic**. |
| `SHORTEST` $k$ `GROUP` | Partitions by endpoints, sorts each partition by path length, groups paths with the same length, then returns all paths in the first $k$ groups from each partition (if fewer than $k$, then all are retained). **Deterministic**. |

**Table 1: Informal semantics of the Selectors defined in GQL.**

| Expression | Informal semantics |
|---|---|
| `WALK` | Is the default option, corresponding to the absence of any filtering. |
| `TRAIL` | Returns paths that do not have any repeated edges. |
| `ACYCLIC` | Returns paths that do not have any repeated nodes. |
| `SIMPLE` | Returns paths with no repeated nodes, except for the first and last node if they are the same. |

**Table 2: Informal semantics of the Restrictors defined in GQL.**

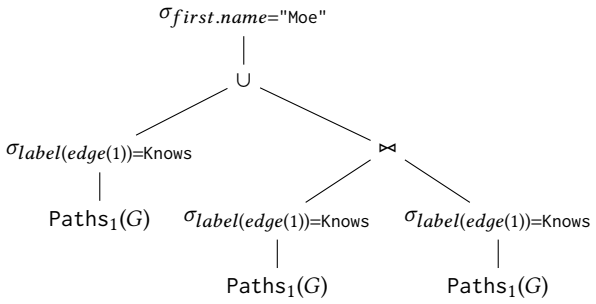



**Figure 3: Example of query tree obtained from a core path algebra expression.**

example, if $p_1 = (n_1, e_1, n_2)$ and $p_2 = (n_2, e_3, n_3)$ then $p_1 \circ p_2 = (n_1, e_1, n_2, e_3, n_3)$.

*Definition 3.1 (Core Path Algebra).* Let $\mathbb{S}$ and $\mathbb{S}'$ be sets of paths and $c$ be a selection condition. The Core Path Algebra is composed by the following operators:

- **Selection:** $\sigma_c(\mathbb{S}) = \{p \in \mathbb{S} \mid ev(p, c) = \texttt{True}\}$
- **Join:** $\mathbb{S} \bowtie \mathbb{S}' = \{p_1 \circ p_2 \mid p_1 \in \mathbb{S} \wedge p_2 \in \mathbb{S}' \wedge \texttt{Last}(p_1) = \texttt{First}(p_2)\}$
- **Union:** $\mathbb{S} \cup \mathbb{S}' = \{p \mid p \in \mathbb{S} \vee p \in \mathbb{S}'\}$

The intuition behind the above definition lies in the path manipulation operators. For instance, a classical operator such as the relational join whose semantics and algorithmic aspects have been widely studied in the database community would not be applicable to paths. Indeed, the conditions on the endpoints of the paths and the returned paths are not easily expressible in a relational setting. Finally, the above core algebra operators correspond to the fundamental operators of Codd's relational algebra [6] under a revised semantics.

## 4 RECURSIVE PATH ALGEBRA

The core algebra described in Section 3 allows us to express fixed-length path queries. In this section, we extend the core algebra with a recursive operator that allows retrieving paths of any length. For example, consider that we want to obtain all the paths from the node `Person` named `"Moe"` to the node `Person` named `"Apu"`, through the label `Knows` one or more times, or through the concatenation of the labels `Likes` and `Has_creator`, zero or more times. This question can be answered by using the following GQL-like query:

```
MATCH p = (x {name:"Moe"})-[(Knows+)|
                (Likes/Has_creator)*]->(y {name:"Apu"})
```

In the above query, `(Knows+)` is a regular expression that allows us to obtain all the paths, of length one or more, that contain edges having the label `Knows`, and connecting the nodes corresponding to `"Moe"` and `"Apu"`. Similarly, the regular expression `(Likes/Has_creator)*` obtains the paths that combine the edge labels `Likes` and `Has_creator`, the one following the other, an undefined number of times (including paths of zero length). This is an example of a recursive regular path query.

Note that a naive evaluation of the above query does not terminate due to the infinite number of solutions produced by the existence of cycles in the graph, particularly those induced by the edges `Knows` and `Has_creator`. This problem can be managed by using a specific path semantics, as we explain below.

First, we define a recursive operator that allows us to evaluate recursive regular path queries and return the entire paths, without any kind of restriction.

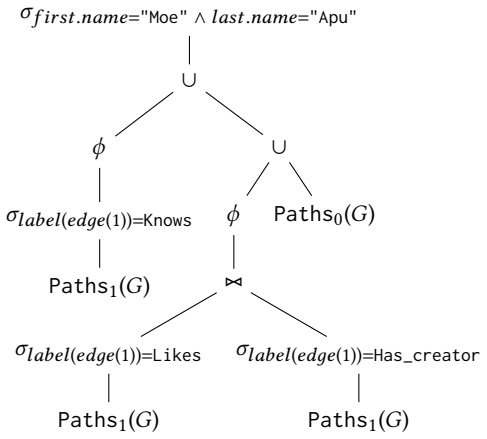



**Figure 4: Evaluation tree of a recursive path algebra query.**

*Definition 4.1 (Recursive operator).* Given a set of paths $\mathbb{S}$, the recursive operator $\phi$ is defined inductively as follows:

(1) Base case: $\phi_0(\mathbb{S}) = \mathbb{S}$.
(2) Recursive case: $\phi_i(\mathbb{S}) = (\phi_{i-1}(\mathbb{S}) \bowtie \phi_0(\mathbb{S})) \cup \phi_{i-1}(\mathbb{S})$ while $i > 0$ and $|\phi_{i-1}| \neq |\phi_i|$.

The recursive operator $\phi(\mathbb{S})$ applies a chain of join operations by using the initial set of paths $\mathbb{S}$ until the fix-point condition is reached. Specifically, in the base case, $\phi_0(\mathbb{S}) = \mathbb{S}$. For the recursion step 1, $\phi_1(\mathbb{S}) = (\phi_0(\mathbb{S}) \bowtie \mathbb{S}) \cup \phi_0(\mathbb{S})$. For the recursion step 2, $\phi_2(\mathbb{S}) = (\phi_1(\mathbb{S}) \bowtie \mathbb{S}) \cup \phi_1(\mathbb{S})$, and so on. The recursion stops in step $n$ when $\phi_n(\mathbb{S})$ is equal to $\phi_{n-1}(\mathbb{S})$, that is, when no new paths have been produced, reaching the fix-point.

For example, Figure 4 shows an evaluation tree that includes the recursive operator $\phi$ twice. In the first case (left branch), the recursive operator receives as input the paths of length one, having the label `Knows`, which were obtained by filtering the set $Paths_1(G)$, that is, all the paths of length one obtained from $G$. Hence, the recursive operator returns the paths containing one or more edges, all of them having the label `Knows`. This is equivalent to the result of the evaluation of the regular expression `(Knows)+`.

In the second case (right branch), the recursive operator returns paths of length 2, 4, 6, etc., such that they repeat the combination of edge labels `Likes` and `Has_creator`. Additionally, the set of paths returned by the recursive operator is united with the set $Paths_0(G)$, which contains the paths of length zero (i.e. the nodes of the graph). The result of such union is equivalent to the semantics of the Kleene star regular expression `(Likes/Has_creator)*`.

It is worth mentioning that the above definition of the recursive operator has a non-halting problem when the input graph contains cycles (see the inner loop formed by the label `Knows` in the graph of Figure 1). In such a case, the recursive operator will never stop, thereby triggering infinite results. This problem can be solved by filtering the paths generated during the recursion, that is, by using a specific semantics for evaluating paths.

Inspired by GQL [18], we defined five versions of the recursive operator, each associated with a specific path semantics, named Walk, Trail, Acyclic, Simple, and Shortest. Given a set of paths $\mathbb{S}$, we introduce the following operators:

- $\phi_{Walk}(\mathbb{S})$, that returns all the paths without any restriction;
- $\phi_{Trail}(\mathbb{S})$, that returns paths without repeated edges;
- $\phi_{Acyclic}(\mathbb{S})$, that returns paths without repeated nodes;
- $\phi_{Simple}(\mathbb{S})$, that returns paths without repeated nodes, with exception of the first and the last node;
- $\phi_{Shortest}(\mathbb{S})$, that returns the paths with the shortest length between the first and the last node.

The formal definition of $\phi_{Walk}(\mathbb{S})$ is given by the inductive method presented in Definition 4.1, that is, the recursive operator without restrictions. The formal definition of $\phi_{Trail}(\mathbb{S})$ extends Definition 4.1 by adding a filter operation in the recursive case. Specifically, the set of paths produced during each recursion step is filtered by eliminating those with repeated edges. The formal definitions of $\phi_{Acyclic}(\mathbb{S})$ and $\phi_{Simple}(\mathbb{S})$ are similar to the definition of $\phi_{Trail}(\mathbb{S})$, but filtering the paths according to the corresponding semantics.

The formal definition of $\phi_{Shortest}(\mathbb{S})$ is also based on Definition 4.1, but is extended to filter the shortest paths. Specifically, for every step $i$, two actions must be applied after computing the set of paths $\phi_i(\mathbb{S})$: (1) for every path $p = (n_s, \ldots, n_t)$ in $\phi_i(\mathbb{S})$, if there is a path $p' = (n'_s, \ldots, n'_t)$ in $\phi_{i-1}(\mathbb{S})$ satisfying that $n_s = n'_s$ and $n_t = n'_t$, then the path $p$ is removed from the set $\phi_i(\mathbb{S})$; (2) if it applies that for every path $p = (n_s, \ldots, n_t)$ in $\phi_0(\mathbb{S})$ there is a path $p' = (n'_s, \ldots, n'_t)$ in $\phi_i(\mathbb{S})$ satisfying that $n_s = n'_s$ and $n_t = n'_t$, then the recursion stops. The first action eliminates paths that are not the shortest paths. The second action stops the recursion when, for every pair of nodes $n_s$ and $n_t$ occurring in the initial set of paths $\phi_0(\mathbb{S})$, we have obtained all the shortest paths between them. Both actions are based on the fact that the paths produced in step $i$ are larger than those produced in step $i-1$.

It is important to mention that each path semantics induces a different set of solutions. For example, given the graph shown in Figure 1 and the regular expression `Knows+`, in Table 3 we show some of the paths returned for each path semantics. Note that, under Walk semantics, there is an infinite number of solutions due to the cycle between nodes $n_2$ and $n_3$.

There are no criteria to say which one of the above recursive operators is the best, but the corresponding semantics have been studied in theory and are implemented by practical graph query languages. Specifically, Gremlin allows arbitrary semantics, SPARQL uses acyclic path semantics, Cypher implements trail semantics, and G-CORE follows the shortest path semantics. GQL supports the above five semantics, even allowing multiple semantics in a single query. Next, we describe how this feature is supported by our algebra.

## 5 EXTENDED ALGEBRA

As described in Section 2.3, GQL and SQL/PGQ introduce path modes (i.e. selectors and restrictors), which allow us to decide which paths are returned and how the paths are computed, respectively. In this section, we provide a formal definition of both concepts and describe their implementation in our algebraic framework.

Consider the following GQL query where `ANY SHORTEST` is the selector keyword and `TRAIL` the restrictor keyword:

```
MATCH ANY SHORTEST TRAIL p = (x)-[Knows]->+(y).
```

According to the (informal) definition for selectors (Table 1) and restrictors (Table 2), the above query is evaluated as follows:

(1) Compute the paths that satisfy the regular expression `Knows+` by complying with the Trail semantics (i.e., paths without repeated edges).
(2) Create groups of paths where each group contains only paths with the same source and target nodes (i.e., the nodes bound to variables $x$ and $y$ respectively).

| ID | Path | W | T | A | S | Sh |
|---|---|---|---|---|---|---|
| $p_1$ | $(n_1, e_1, n_2)$ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $p_2$ | $(n_1, e_1, n_2, e_2, n_3, e_3, n_2)$ | ✓ | ✓ | | | |
| $p_3$ | $(n_1, e_1, n_2, e_2, n_3)$ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $p_4$ | $(n_1, e_1, n_2, e_2, n_3, e_3, n_2, e_2, n_3)$ | ✓ | | | | |
| $p_5$ | $(n_1, e_1, n_2, e_4, n_4)$ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $p_6$ | $(n_1, e_1, n_2, e_2, n_3, e_3, n_2, e_4, n_4)$ | ✓ | ✓ | | | |
| $p_7$ | $(n_2, e_2, n_3, e_3, n_2)$ | ✓ | ✓ | | ✓ | ✓ |
| $p_8$ | $(n_2, e_2, n_3, e_3, n_2, e_2, n_3, e_3, n_2)$ | ✓ | | | | |
| $p_9$ | $(n_2, e_2, n_3)$ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $p_{10}$ | $(n_2, e_2, n_3, e_3, n_2, e_2, n_3)$ | ✓ | | | | |
| $p_{11}$ | $(n_2, e_4, n_4)$ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $p_{12}$ | $(n_2, e_2, n_3, e_3, n_2, e_4, n_4)$ | ✓ | ✓ | | | |
| $p_{13}$ | $(n_3, e_3, n_2, e_4, n_4)$ | ✓ | ✓ | ✓ | ✓ | ✓ |
| $p_{14}$ | $(n_3, e_3, n_2, e_2, n_3, e_3, n_2, e_4, n_4)$ | ✓ | | | | |

**Table 3: Given the graph shown in Figure 1, this table shows some (of an infinite number of) paths satisfying the regular expression Knows+, this under different semantics (W*alk*, T*rail*, A*cyclic*, S*imple* and Sh*ortest*).**

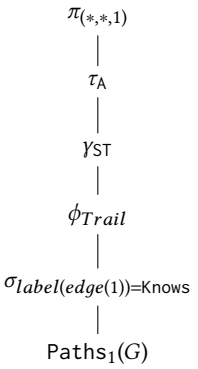



**Figure 5: A query plan including order-by, group-by and projection.**

(3) Pick a single *shortest* path (`ANY SHORTEST`) from each group, randomly (recall that there might be several trails of the same shortest length in each group).

Hence, the restrictor allows us to define the path semantics used to compute the paths, and the selector allows us to filter out the resulting paths.

To support the query mentioned above, we introduced three algebraic operators: group-by ($\gamma$), order-by ($\tau$) and projection ($\pi$). To explain these operators, we will use the evaluation tree shown in Figure 5, which emulates the GQL query described above. In the following, we enumerate the steps from the leaf to the root of the algebraic tree.

**Step 1.** The operator $\mathsf{Paths}_1(G)$ returns the paths of length one (i.e., the edges in $G$).

**Step 2.** The selection operator $\sigma$ filters the edges to those having label Knows (i.e. $e_1$, $e_2$, $e_3$ and $e_4$).

**Step 3.** The recursive operator $\phi_{Trail}$ computes the paths that satisfy the regular expression Knows+ applying the trail semantics. It results in the set of paths $\{p_1, p_2, p_3, p_5, p_6, p_7, p_9, p_{11}, p_{12}, p_{13}\}$ shown in Table 3 (column **T**).

| Group-by expression | Solution space organization |
|---|---|
| $\gamma$ | 1 partition, 1 group |
| $\gamma_{\mathsf{S}}$ | $N$ partitions, 1 group per partition |
| $\gamma_{\mathsf{T}}$ | $N$ partitions, 1 group per partition |
| $\gamma_{\mathsf{L}}$ | 1 partition, $M$ groups per partition |
| $\gamma_{\mathsf{ST}}$ | $N$ partitions, 1 group per partition |
| $\gamma_{\mathsf{SL}}$ | $N$ partitions, $M$ groups per partition |
| $\gamma_{\mathsf{TL}}$ | $N$ partitions, $M$ groups per partition |
| $\gamma_{\mathsf{STL}}$ | $N$ partitions, $M$ groups per partition |

**Table 4: Group-by expressions and the corresponding solution space organizations. Note that $N > 0$ and $M > 0$.**

| Partition $P$ | Group $G$ | Path $p$ | MinL($P$) | MinL($G$) | Len($p$) |
|---|---|---|---|---|---|
| $part_1$ | $group_{11}$ | $p_1$ | 1 | 1 | 1 |
| | | $p_2$ | | | 3 |
| $part_2$ | $group_{21}$ | $p_3$ | 1 | 1 | 1 |
| $part_3$ | $group_{31}$ | $p_5$ | 2 | 2 | 2 |
| | | $p_6$ | | | 4 |
| $part_4$ | $group_{41}$ | $p_7$ | 2 | 2 | 2 |
| $part_5$ | $group_{51}$ | $p_9$ | 1 | 1 | 1 |
| $part_6$ | $group_{61}$ | $p_{11}$ | 1 | 1 | 1 |
| | | $p_{12}$ | | | 3 |
| $part_7$ | $group_{71}$ | $p_{13}$ | 2 | 2 | 2 |

**Table 5: Example of solution space produced by the group-by operator $\gamma_{\mathsf{ST}}$. The parameter ST implies many partitions, one group per partition, and many paths per group.**

**Step 4.** The group-by operator $\gamma_{\mathsf{ST}}$ transforms the above set of paths into a solution space, a special data structure composed of partitions and groups. Specifically, a solution space is composed of one or more partitions, each partition is composed of one or more groups, and each group contains one or more paths. A partition organizes the paths based on their endpoints (i.e. the source and target nodes of a path), and a group organize the paths based on their length. Hence, a given combination of *Source*, *Target* and *Length* induces a solution space with a specific organization of partitions and groups as shown in Table 4.

Recalling our example, the operator $\gamma_{\mathsf{ST}}$ (*Source-Target*) implies a solution space with $N$ partitions, where each partition contains a single group with the paths having the same source and target nodes. Hence, there will be one group for each pair of people connected by a path satisfying the regular expression knows+. A tabular representation of this solution space is shown in Table 5.

**Step 5.** The order-by operator ($\tau_\theta$) allows us to sort the partitions, the groups and the paths composing a solution space. The parameter $\theta$ indicates the ordering criterion, allowing the values P, G, A, PG, PA, GA and PGA. Specifically, $\tau_{\mathsf{P}}$ (*order-by partition*) means that the partitions are sorted by the length of their shortest path, in ascending order; $\tau_{\mathsf{G}}$ (*order-by group*) means that the groups inside each partition are sorted by the length of their shortest path; $\tau_{\mathsf{A}}$ (*order-by path*) means that the paths inside each group are sorted by length; $\tau_{\mathsf{PG}}$ (*order-by partition-group*) sorts by both, partition and group; similarly for the remaining cases.

Following our example, the operation $\tau_{\mathsf{A}}$ sorts the paths inside each group of the solution space produced by the group-by operator. It can be observed in Table 5, where the column MinL($P$) indicates the length of the shortest path in a partition $P$, MinL($G$) indicates the length of the shortest path in a group $G$, and Len($p$)

indicates the length of a path $p$. Note that these columns can be used to sort partitions, groups and paths.

**Step 6**. The projection operator ($\pi$) allows us to transform a solution space into a set of paths. To do this, the projection operator receives as parameter a tuple of the form $(\#_P, \#_G, \#_A)$ where each # can be either the symbol $*$ or a positive integer. Hence, $\#_P$ indicates the number of partitions to be returned, $\#_G$ indicates the number of groups (per partition) to be returned, and $\#_A$ indicates the number of paths (per group) to be returned.

Returning to our example, the expression $\pi_{(*,*,1)}$ returns one path per group (the first one in the group; that is, the shortest one since in the previous step we sorted the paths by length), for every group inside each partition. Hence, the final output of the query will be the set of paths $\{p_1, p_3, p_5, p_7, p_9, p_{11}, p_{13}\}$.

In the remainder of this section, we study the formal semantics for the algebraic operators group-by, order-by and project, all of which use the notion of solution space.

*Definition 5.1.* A *Solution Space* is a tuple $SS = (\mathbb{S}, \mathbb{G}, \mathbb{P}, \alpha, \beta, \triangle)$ where: $\mathbb{S}$ is a set of paths; $\mathbb{P}$ is a set of partitions; $\mathbb{G}$ is a set of groups; $\alpha : \mathbb{S} \rightarrow \mathbb{G}$ is a total function that assigns each path to a group; $\beta : \mathbb{G} \rightarrow \mathbb{P}$ is a total function that assigns each group to a partition; and $\triangle : (\mathbb{S} \cup \mathbb{G} \cup \mathbb{P}) \rightarrow \mathbb{Z}^+$ is a total function used to assign a positive integer to paths, groups and partitions.

The concept of a solution space involves a data structure that organizes a set of paths into groups (function $\alpha$), which are further organized into partitions (function $\beta$). In addition, the function $\triangle$ is used to sort the elements within the solution space. For example, assume that $x$, $y$, and $z$ are paths within a group $g$. If $\triangle(x) = 1$, $\triangle(y) = 2$ and $\triangle(z) = 3$, then we establish a virtual order of the paths inside $g$. On the other hand, if the three paths have the same value for $\triangle$, then there is no order among them. The same approach is used to sort groups within a partition and to sort the partitions within the solution space.

## 5.1 Group by

Given a set of paths $\mathbb{S}$, the group-by operator is represented as $\gamma_\psi(\mathbb{S})$ where $\psi \in \{\emptyset, \mathtt{S}, \mathtt{T}, \mathtt{L}, \mathtt{ST}, \mathtt{SL}, \mathtt{TL}, \mathtt{STL}\}$ ($\mathtt{S}$ = Source, $\mathtt{T}$ = Target, $\mathtt{L}$ = Length, $\mathtt{ST}$ = Source-Target, $\mathtt{SL}$ = Source-Length, $\mathtt{TL}$ = Target-Length, $\mathtt{STL}$ = Source-Target-Length). The evaluation of $\gamma_\psi(\mathbb{S})$ returns a solution space $SS = (\mathbb{S}, \mathbb{G}, \mathbb{P}, \alpha, \beta, \triangle)$ defined as follows:

- If $\psi$ is $\emptyset$ then $\mathbb{P} = \{P_1\}$, $\mathbb{G} = \{G_1\}$, $\beta(G_1) = P_1$, and $\forall p \in \mathbb{S}$ it applies that $\alpha(p) = G_1$.
- If $\psi$ is $\mathtt{S}$ then $\forall s \in \{\mathtt{First}(p) \mid p \in \mathbb{S}\}$ there will be a partition $P_s = \{G_s\}$ where $G_s = \{p' \in \mathbb{S} \mid \mathtt{First}(p') = s\}$.
- If $\psi$ is $\mathtt{T}$ then $\forall t \in \{\mathtt{Last}(p) \mid p \in \mathbb{S}\}$ there will be a partition $P_t = \{G_t\}$ where $G_t = \{p' \in \mathbb{S} \mid \mathtt{Last}(p') = t\}$.
- If $\psi$ is $\mathtt{L}$ then $\mathbb{P} = \{P_1\}$, $\forall l \in \{\mathtt{Len}(p) \mid p \in \mathbb{S}\}$ there will be a group $G_l = \{p' \in \mathbb{S} \mid \mathtt{Len}(p') = l\}$ with $\beta(G_l) = P_1$.
- If $\psi$ is $\mathtt{ST}$ then $\forall (s,t) \in \{(\mathtt{First}(p), \mathtt{Last}(p)) \mid p \in \mathbb{S}\}$ there will be a partition $P_{st} = \{G_{st}\}$ where $G_{st} = \{p' \in S \mid \mathtt{First}(p') = s \wedge \mathtt{Last}(p') = t\}$.
- If $\psi$ is $\mathtt{SL}$ then $\forall s \in \{\mathtt{First}(p) \mid p \in \mathbb{S}\}$ there will be a partition $P_s$, and $\forall l \in \{\mathtt{Len}(p) \mid p \in \mathbb{S} \wedge \mathtt{First}(p) = s\}$ there will be a graph $G_{sl} = \{p' \in \mathbb{S} \mid \mathtt{First}(p') = s \wedge \mathtt{Len}(p') = l\}$ with $\beta(G_{sl}) = P_s$.
- If $\psi$ is $\mathtt{TL}$ then $\forall t \in \{\mathtt{Last}(p) \mid p \in \mathbb{S}\}$ there will be a partition $P_t$, and $\forall l \in \{\mathtt{Len}(p) \mid p \in \mathbb{S} \wedge \mathtt{Last}(p) = t\}$ there will be a graph $G_{tl} = \{p' \in \mathbb{S} \mid \mathtt{Last}(p') = t \wedge \mathtt{Len}(p') = l\}$ with $\beta(G_{tl}) = P_t$.
- If $\psi$ is $\mathtt{STL}$ then $\forall (s,t) \in \{(\mathtt{First}(p), \mathtt{Last}(p)) \mid p \in \mathbb{S}\}$ there will be a partition $P_{st}$, and $\forall l \in \{\mathtt{Len}(p) \mid p \in S \wedge \mathtt{First}(p) = s \wedge \mathtt{Last}(p) = t\}$ there will be a graph $G_{stl} = \{p' \in \mathbb{S} \mid \mathtt{First}(p') = s \wedge \mathtt{Last}(p') = t \wedge \mathtt{Len}(p') = l\}$ with $\beta(G_{stl}) = P_{st}$.

| $\theta$ | $\forall P \in \mathbb{P}$ | $\forall G \in \mathbb{G}$ | $\forall p \in \mathbb{S}$ |
|---|---|---|---|
| P | $\triangle'(P) = \mathtt{MinL}(P)$ | $\triangle'(G) = \triangle(G)$ | $\triangle'(p) = \triangle(p)$ |
| G | $\triangle'(P) = \triangle(P)$ | $\triangle'(G) = \mathtt{MinL}(G)$ | $\triangle'(p) = \triangle(p)$ |
| A | $\triangle'(P) = \triangle(P)$ | $\triangle'(G) = \triangle(G)$ | $\triangle'(p) = \mathtt{Len}(p)$ |
| PG | $\triangle'(P) = \mathtt{MinL}(P)$ | $\triangle'(G) = \mathtt{MinL}(G)$ | $\triangle'(p) = \triangle(p)$ |
| PA | $\triangle'(P) = \mathtt{MinL}(P)$ | $\triangle'(G) = \triangle(G)$ | $\triangle'(p) = \mathtt{Len}(p)$ |
| GA | $\triangle'(P) = \triangle(P)$ | $\triangle'(G) = \mathtt{MinL}(G)$ | $\triangle'(p) = \mathtt{Len}(p)$ |
| PGA | $\triangle'(P) = \mathtt{MinL}(P)$ | $\triangle'(G) = \mathtt{MinL}(G)$ | $\triangle'(p) = \mathtt{Len}(p)$ |

**Table 6: Semantics of the order-by operator $\tau_\theta$. Given a solution space $SS = (\mathbb{S}, \mathbb{G}, \mathbb{P}, \alpha, \beta, \triangle)$, the evaluation of $\tau_\theta(SS)$ returns a solution space $SS' = (\mathbb{S}, \mathbb{G}, \mathbb{P}, \alpha, \beta, \triangle')$ where function $\triangle'$ is the only change. For each value of the parameter $\theta$, this table shows the assignments for function $\triangle'$.**

Furthermore, it applies that: $\triangle(p) = 1$ for every path $p \in \mathbb{S}$, $\triangle(G) = 1$ for every group $G \in \mathbb{G}$, and $\triangle(P) = 1$ for every partition $P \in \mathbb{P}$. Recall that function $\triangle$ can be used to introduce a virtual order of the elements (paths, groups and partitions) within a solution space. In this case, there is no such order as all elements have the same value for $\triangle$.

## 5.2 Order by

Let $SS = (\mathbb{S}, \mathbb{G}, \mathbb{P}, \alpha, \beta, \triangle)$ be a solution space. Given a group $G \in \mathbb{G}$, we will use the function $\mathtt{MinL}(G)$ to obtain the length of the shortest path in $G$. Similarly, for a given partition $P \in \mathbb{P}$, the function $\mathtt{MinL}(P)$ returns the minimum length among all the groups in $P$.

Given a solution space $SS$, the order-by operator is represented as $\tau_\theta(SS)$ where $\theta \in \{\mathtt{P}, \mathtt{G}, \mathtt{A}, \mathtt{PG}, \mathtt{PA}, \mathtt{GA}, \mathtt{PGA}\}$ ($\mathtt{P}$ = Partition, $\mathtt{G}$ = Group, $\mathtt{A}$ = Path, $\mathtt{PG}$ = Partition-Group, $\mathtt{PA}$ = Partition-Path, $\mathtt{GA}$ = Group-Path, $\mathtt{PGA}$ = Partition-Group-Path). The evaluation semantics of $\tau_\theta(SS)$ is presented in Table 6. Given a solution space $SS = (\mathbb{S}, \mathbb{G}, \mathbb{P}, \alpha, \beta, \triangle)$, the operator $\tau_\theta(SS)$ redefines the function $\triangle$ to a new function $\triangle'$ depending on the parameter $\theta$. For example, if $\theta$ is $\mathtt{P}$ then, for each partition $P$ it applies that $\triangle'(P) = \mathtt{MinL}(P)$, for each group $G$ in $P$ it applies that $\triangle'(G) = \triangle(G)$, and for each path $p$ in $G$ it applies that $\triangle'(p) = \triangle(p)$.

Note that the order-by operator uses the function $\triangle'$ to introduce a virtual ordering of paths, groups and partitions. The order of a path, inside a group, is given by its length. The order of a group, inside a partition, is given by the length of its shortest path. The order of a partition, inside a solution space, is given by the length of the shortest path contained in its groups.

## 5.3 Projection

Given a solution space $SS = (\mathbb{S}, \mathbb{G}, \mathbb{P}, \alpha, \beta, \triangle)$, the projection operator is represented as $\pi_{(\#_P, \#_G, \#_A)}(SS)$ where each # is either the symbol $*$ or a positive integer. Here, $\#_P$ indicates the number of partitions to be projected, $\#_G$ indicates the number of groups (per partition) to be projected, and $\#_A$ indicates the number of paths (per group) to be projected.

The evaluation of $\pi_{(\#_P, \#_G, \#_A)}(SS)$ returns a set of paths according to Algorithm 1. The final set of paths will be stored in the

variable $\mathbb{S}_{out}$. First, the algorithm transforms the set of partitions $\mathbb{P}$ into the sequence *SeqP* where the partitions are sorted in ascending order according to their length defined by function $\triangle$ (line 2). The number of partitions to be processed is defined in variable *maxP* according to the parameter $\#_P$ (lines 3 and 4). For each partition $P$ (up to processing of partitions *maxP*, the algorithm creates a sequence *SeqG* that contains the groups of $P$ (lines 6 to 8). The groups in *SeqG* are sorted in ascending order according to the function $\triangle$. The variable *maxG* indicates the number of groups to be processed according to the parameter $\#_G$ (lines 9 and 10). For each group $G$ (up to processing groups *maxG*, the algorithm gets the paths of $G$ and creates the sequence *SeqS*, where the paths are sorted according to their length (lines 12 to 14). The variable *maxS* indicates the number of paths to be processed according to the parameter $\#_A$ (lines 15 and 16). For each path $p$ (up to processing of *maxS* paths), the algorithm adds $p$ to the final set of paths $\mathbb{S}_{out}$ (lines 17 to 20).

Note that the procedure shown in Algorithm 1 can be improved in several ways: the *Sort* function can implement a special sorting algorithm; indexes can be used to facilitate the retrieval of paths inside a group, as well as the groups inside a partition; the calls to ordering functions (lines 2, 8 and 14) are unnecessary when the input solution space comes from the group-by operator (i.e. the query does not include the order-by operator). Moreover, Algorithm 1 can be easily extended to support the projection of partitions, groups, and paths in descending order.

```
   input : A solution space SS = (𝕊, 𝔾, ℙ, α, β, △) and the
           projection parameters #P, #G and #A.
   output: A set of paths 𝕊out.
1  𝕊out ← ∅;
2  SeqP ← Sort(ℙ); //Sort ℙ based on function △
3  if #P is "*" ∨ #P > length(SeqP) then
     maxP ← length(SeqP);
4  else maxP ← #P;
5  for i ← 1 to maxP do
6      P ← SeqP[i];
7      SetG ← {G ∈ 𝔾 | β(G) = P}; //Get the groups of P
8      SeqG ← Sort(SetG); //Sort the groups of P based on △
9      if #G is "*" ∨ #G > length(SeqG) then
         maxG ← length(SeqG);
10     else maxG ← #G;
11     for j ← 1 to maxG do
12         G ← SeqG[j];
13         SetS ← {p ∈ S | α(p) = G}; //Get the paths of G
14         SeqS ← Sort(SetS); //Sort the paths of G based on
             △
15         if #A is "*" ∨ #A > length(SeqS) then
             maxS ← length(SeqS);
16         else maxS ← #A;
17         for k ← 1 to maxS do
18             p ← SeqS[k];
19             Add(p, 𝕊out) //Add the path p to the set Sout
20         end
21     end
22 end
```

**Algorithm 1:** Projection function $\pi_{(\#_P,\#_G,\#_A)}(SS)$

| GQL expression | Path algebra expression |
|---|---|
| `ALL WALK` *ppe* | $\pi_{(*,*,*)}(\gamma(\phi_{Walk}(\mathsf{RE})))$ |
| `ANY SHORTEST WALK` *ppe* | $\pi_{(*,*,1)}(\tau_{\mathsf{A}}(\gamma_{\mathsf{ST}}(\phi_{Walk}(\mathsf{RE}))))$ |
| `ALL SHORTEST WALK` *ppe* | $\pi_{(*,1,*)}(\tau_{\mathsf{G}}(\gamma_{\mathsf{STL}}(\phi_{Walk}(\mathsf{RE}))))$ |
| `ANY WALK` *ppe* | $\pi_{(*,*,1)}(\gamma_{\mathsf{ST}}(\phi_{Walk}(\mathsf{RE})))$ |
| `ANY` $k$ `WALK` *ppe* | $\pi_{(*,*,k)}(\gamma_{\mathsf{ST}}(\phi_{Walk}(\mathsf{RE})))$ |
| `SHORTEST` $k$ `WALK` *ppe* | $\pi_{(*,*,k)}(\tau_{\mathsf{A}}(\gamma_{\mathsf{ST}}(\phi_{Walk}(\mathsf{RE}))))$ |
| `SHORTEST` $k$ `GROUP WALK` *ppe* | $\pi_{(*,k,*)}(\tau_{\mathsf{G}}(\gamma_{\mathsf{STL}}(\phi_{Walk}(\mathsf{RE}))))$ |

**Table 7: For each GQL selector-restrictor expression, this table shows the corresponding path algebra expression.**

## 6 COMPARISON WITH GQL

Previously, we have described the seven types of selectors and the four types of restrictors supported by GQL. According to the GQL specification, it is possible to combine every selector with every restrictor, resulting in 28 combinations. An important fact is that every selector-restrictor combination can be translated into a path algebra expression, whose evaluation satisfies the informal semantics defined in Table 1 and Table 2.

In Table 7, we present the GQL expressions produced by combining the seven selectors with the restrictor `WALK`, and show the corresponding path algebra expressions. Note that: `RE` denotes the regular expression obtained from the path pattern expression *ppe*; every selector (e.g. `ALL`) is translated to an expression containing the group-by ($\gamma$), order-by ($\tau$) and projection ($\pi$) operators; and the restrictor `WALK` is translated to the recursive operator $\phi_{Walk}$. The path algebra expressions shown in Table 7, can be replicated to translate the rest of restrictors by just replacing the term `WALK` with `TRAIL`, `ACYCLIC` or `SIMPLE`.

For instance, the GQL expression

```
MATCH ALL SHORTEST ACYCLIC p = (x)-[Knows]->+(y)
```

can be translated to the path algebra expression

$$\pi_{(*,1,*)}(\tau_{\mathsf{G}}(\gamma_{\mathsf{STL}}(\phi_{Acyclic}(\sigma_{label(edge(1))=\mathtt{Knows}}(\mathtt{Paths}_1(G))))))$$

where: $\phi_{Acyclic}$ returns a set of paths where each path $p$ satisfies the regular expression `Knows+` and $p$ does not have any repeated nodes; $\gamma_{\mathsf{STL}}$ transforms the set of paths into a solution space where, for each source-target combination there is a partition $P_i$, and each partition $P_i$ contains a group $G_{ij}$ containing the paths with the same length; $\tau_{\mathsf{G}}$ sorts the groups inside each partition, in increasing order, according to their length (in this case, the length of each group is given by the length of the shortest path inside it); and $\pi_{(*,1,*)}$ returns the paths contained in the first group of each partition (i.e. all the shortest paths for each source-target combination).

On the other hand, there exist path algebra expressions that are not supported by GQL. It is given by the 8 types of group-by, 7 types of order-by, the 7 types of projection, and the 5 types of recursion, which results in 1960 combinations, surpassing the 28 combinations defined by GQL. For instance, the following algebra expression is not supported by GQL:

$$\pi_{(*,*,1)}(\tau_{\mathsf{G}}(\gamma_{\mathsf{L}}(\phi_{Trail}(\sigma_{label(edge(1))=\mathtt{Knows}}(\mathtt{Paths}_1(G)))))).$$

Note that $\phi_{Trail}$ computes the trails satisfying `Knows+`; $\gamma_{\mathsf{L}}$ creates a solution space with a single partition and many groups where each group contains the paths with the same length; $\tau_{\mathsf{G}}$ sort the groups in ascending order according to their length (i.e., the length of a group is equal to the length of the paths contained in the group); and $\pi_{(*,*,1)}$ returns a single path from each group.

Therefore, the above algebra expression returns a sample trail of each possible length.

Another issue is the practical use of some path algebra expressions. For instance, the expression

$$\pi_{(*,*,1)}(\tau_{\mathsf{PG}}(\gamma(\phi_{Walk}(\sigma_{label(edge(1))=\mathtt{Knows}}(\mathtt{Paths}_1(G))))))$$

is somehow redundant and unnecessarily complex as we just like to return a single path (Non-Deterministic), without imposing any condition. Note that the order-by operator $\tau_{\mathsf{PG}}$ is unnecessary, as operator $\gamma$ returns a solution space with a single partition and a single group. Hence, the above expression can be replaced by

$$\pi_{(*,*,1)}(\gamma(\phi_{Trail}(\sigma_{label(edge(1))=\mathtt{Knows}}(\mathtt{Paths}_1(G))))).$$

For the sake of space, we are not presenting a complete analysis of the equivalences among the path expressions supported by our path algebra and GQL. However, the above examples give an intuition of the expressive power of our algebra, and show that it supports more types of queries that the ones supported by GQL. In terms of completeness, our algebra allows for all possible combinations of its operators, leaving the responsibility of using them correctly to the user. This approach is also followed in practical query languages like SQL and Cypher.

# 7 CONNECTING THE PATH ALGEBRA WITH GQL

Our main motivation to define a path algebra is to allow formulating logical plans for the evaluation of path queries in a fashion similar to the usage of relational algebra for evaluating SQL queries. Most notably, having a path algebra at our disposal allows for a quick formulation of logical plans in any engine wishing to support queries that involve complex conditions over paths. To aid with this process, we extended the GQL syntax to support all the features provided by the algebra, and implemented a query parser for it.

## 7.1 GQL Extension

According to the GQL specification, the structure of a path query is given by the following grammar:

```
<pathQuery> ::= MATCH <selector> <restrictor>
                <pathPattern>
<selector> ::= ALL | ANY SHORTEST | ALL SHORTEST |
               ANY | ANY <number> |
               SHORTEST <int> | SHORTEST <int> GROUP
<restrictor> ::= WALK | TRAIL | SIMPLE | ACYCLIC
<pathPattern> ::= <var> = <pathExp> WHERE <condition>
```

where <pathExp> is an expression of the form (var)-[RegExp]->(var), and <condition> is a selection condition.

In order to support the operators of our path algebra, we propose to modify the above structure as follows:

```
<pathQuery> ::= MATCH <projection> <restrictor_ext>
                <pathPattern> <groupby>? <orderby>?
<projection> ::= <partProj> <groupProj> <pathProj>
<partProj> ::= ( ALL | <number> ) PARTITIONS
<groupProj> ::= ( ALL | <number> ) GROUPS
<pathProj> ::= ( ALL | <number> ) PATHS
<restrictor_ext> ::= WALK | TRAIL | SIMPLE | ACYCLIC | SHORTEST
<groupby> ::= (SOURCE)? (TARGET)? (LENGTH)?
<orderby> ::= (PARTITION)? (GROUP)? (PATH)?
```

Hence, if we want to compute all the trails for each pair of nodes in the graph, and return a single path for each target node, we can use the query

```
MATCH ALL PARTITIONS ALL GROUPS 1 PATHS
TRAIL p = (x)-[(Knows)*]->(y)
GROUP BY TARGET ORDER BY PATH
```

which corresponds to the path algebra expression

$$\pi_{(*,*,1)}(\tau_{\mathsf{A}}(\gamma_{\mathsf{T}}(\phi_{Trail}(\sigma_{label(edge(1))=\mathtt{Knows}}(\mathtt{Paths}_1(G)))))).$$

## 7.2 Query Parser

We have developed[4] an open-source parser that transforms declarative queries into logical query plans. Our parser is a Java application with two main components: the query parser and the logical plan generator. The query parser reads a path query expression and returns a parse tree. This is achieved by using the *ANTLR*[5] library. The logical plan generator traverses the parse tree, extracting all the algebraic operations, and generates a query tree.

The query parser has a command-line interface where users can write a path query expression and get the corresponding parse tree. For example, if we input the sample query shown above in this section, the parser generates the following query plan:

```
Projection (ALL PARTITIONS ALL GROUPS 1 PATHS)
OrderBy (Path)
Group (Target)
Restrictor (TRAIL)
-> Recursive Join (restrictor: TRAIL)
 -> Select: (label(edge(1)) = Knows , Paths(G,1))
```

The output of the parser displays the query plan as a textual tree. The initial lines (1 to 4) display the selected parameters for the projection, order by, group by, and restrictor statements. Subsequently, lines 5 and 6 present the query, with indentation indicating the depth of each instruction and its corresponding branch.

Notice that our logical plans pave the way for building implementations of GQL and SQL/PGQ standards, or for general engines wishing to support path queries. Namely, to build a reference implementation, one only needs to specify an algorithm for each operator of the algebra, as these suffice to define any path query in the two standards (and some additional ones as well). Notice that algorithms for specific algebra operations we support are independent research topics of their own [8, 32, 37], so we find building such a reference implementation to be outside of the scope of the current paper.

## 7.3 Query optimization

A well-know advantage of having a query algebra is that it facilitates query optimization. In particular, the query engine can perform logical optimizations (e.g., predicate pushdown, column pruning) and physical optimizations (e.g., better join strategies).

The classic example of logical optimization is pushing filters [14]. For example, the query plan shown in Figure 6a can be optimized by pushing down the selection $\sigma_{first.name=\mathtt{"Moe"}}$. The optimized query plan is shown in Figure 6b. Note that this change allows us to reduce the number of intermediate results (paths) in advance, and consequently, to reduce the number of join comparisons.

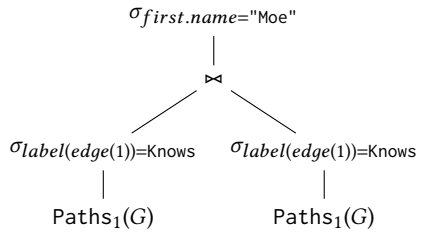
The introduction of selectors and restrictors have opened the door to new types of rewritings. For instance, the expression

$$\pi_{(1,1,*)}(\tau_{\mathsf{G}}(\gamma_{\mathsf{L}}(\phi_{Walk}(\sigma_{label(edge(1))=\mathtt{Knows}}(\mathtt{Paths}_1(G))))))$$

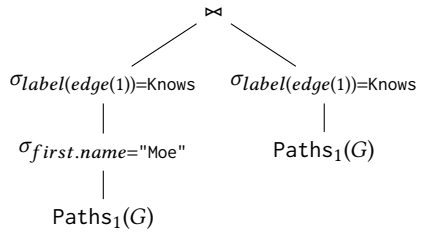
can be used to obtain the shortest paths that satisfy the regular expression Knows+. However, this expression just works well when the target graph does not contain cycles for the edges labeled

[4] https://github.com/pathalgebra/AlgebraParser

[5] ANTLR (ANother Tool for Language Recognition), https://www.antlr.org




**(a) A basic query plan.**




**(b) A basic optimized query plan.**

**Figure 6: A basic optimized query plan for a query with basic algebra expression.**

with Knows. The equivalence also applies if the implementation of $\phi_{Walk}$ limits the recursion to a given length of paths, long enough to return all the shortest paths. Otherwise, equivalence does not apply. Instead of the above, the optimization engine can produce the algebra expression

$$\pi_{(1,1,*)}(\gamma(\phi_{Shortest}(\sigma_{label(edge(1))=\text{Knows}}(\text{Paths}_1(G)))))$$

where the recursive operator and the group-by operator have been changed, and the order-by operator has been removed. The change of $\phi_{Walk}$ by $\phi_{Shortest}$ is very important because now the query returns a finite number of solutions, i.e. it always terminates.

Overall, such manipulations are a standard part of any cost-based query execution plan in SQL databases, and have a high potential to be used over graphs. Indeed, some engines already considered such optimizations for restricted versions of path algebra, or for simply detecting, but not returning paths [34, 37], and there is a rich body of literature on query rewriting for path queries [5].

## 8 RELATED WORK

Since the pioneering work of Edgar Codd [6], relational algebra has been a hallmark of data management research focusing on relational query processing and optimization. Despite the fact that graph databases have established themselves as a data-driven technology with high demand in industry, a counterpart of relational algebra is absent in graph database research. In this paper, we fill this gap and propose a path-based algebraic framework, that seamlessly work for recursive graph queries.

In the following, we discuss related work in this area, focusing primarily on extensions of relational algebra to support path queries, algorithmic approaches for computing path queries, and the methods used by current database systems.

### 8.1 Extensions of relational algebra

First, we review extensions of relational algebra that support the evaluation of path queries. $\mu$-RA [19] extends the relational algebra with a recursive operator $\mu$ that enables the recursive computation of joins. Although this extension can be used to simulate the evaluation of path queries, there is no way to implement different path semantics.

Francis et al. [11] proposed GPC, a declarative non-procedural calculus for property graphs similar in spirit to TRC (tuple-relational calculus) for relational data. Graph patterns in GPC generalize conjunctive two-way regular path queries [4] to property graphs. Despite supporting restrictors among the set of simple, trail (used by default if none is given), and shortest, a GPC query returns tuples containing assignments of variables to values. Hence, GPC does not manipulate paths as we propose in our algebraic framework.

There exist early attempts to define non-recursive algebras for property graph queries [4, 13, 27]. All these attempts closely resemble relational algebra as they redefine selection, node-based join, edge-based join, and union in a similar fashion and add a path navigation operator, which is meant to encode linear recursion in RPQ. They disregard the definition of a path-oriented algebraic framework equipped with a projection operator and a solution space allowing for full expressiveness, covering recent standard graph query languages and beyond. Moreover, they disregard several features of standard graph query languages, such as restrictors and selectors, and do not map to concrete and practical standard query languages.

There exist other pieces of work where the notion of path algebras is used, such as case studies of graph problems involving paths [15] (connectivity, shortest path, path enumeration, among others). The search for paths between a pair of nodes using operators for paths such as join and product has also been studied [23], along with recursive operators for paths with their evaluation based on automata [29]. However, these studies are prior to the appearance of property graph-related languages and are tailored to plain labeled graphs and non-recursive queries.

It is worth mentioning that there is a substantial body of work on processing SPARQL property path using an algebraic approach. Some of these [22, 28, 37] include defining an intermediate algebra that supports recursion. However, apart from the inherent differences between RDF and property graphs, they are not considering how paths should be returned, since this is not supported in SPARQL, and they do not allow any path mode besides ANY WALK.

### 8.2 Algorithmic approaches for computing path queries

The literature presents several algorithmic approaches for computing path queries in graph databases. The most basic approach is to extend a graph traversal algorithm, such as depth-first search or breadth-first search, by incorporating regular expression matching during the traversal. This approach can be improved by using several techniques, including parallelism [26], approximation [35], distributed processing [16], and compact structures [3].

Automata-based approaches traverse the graph while tracking the states of an automaton constructed from the regular expression, as a set of transitions maps directly to paths in the graph [25]. Index-based approaches precompute and index paths based on their edge labels or combinations of labels, which reduces the search space and can accelerate path computation [9, 20]. Matrix-based methods represent the graph as an adjacency matrix, enabling the use of matrix multiplication to find paths between nodes [3].

Works such as [8, 24] describe specialized algorithms for executing a single path query, or compressing the paths in the result set, but they do not discuss on how these solutions can be incorporated into a larger query pipeline, nor how to specify an algebra to manipulate the output paths, unlike our approach.

These algorithmic approaches vary in efficiency, scalability, and complexity, and are often selected based on the specific characteristics of the graph and the query workload. For instance, automata-based approaches are particularly effective for graphs with clear label patterns, while index-based and matrix-based methods tend to be more efficient for large-scale graphs.

An important disadvantage of using an algorithm is that we are not able to apply query optimization techniques. For doing so, we need a query algebra like the one described in [37] and [19]. A common issue of these algebras is that they do not return the entire paths, just the source and target nodes for each path.

### 8.3 Path query evaluation in current database systems

We conducted a brief review of current database systems to understand the methods used for evaluating path queries. Our revision included the following systems: Amazon Neptune, ArangoDB, DuckDB, GraphScope, MemGraph, MillenniumDB, NebulaGraph, Neo4j, OraclePGX, OrientDB, RedisGraph, TigerGraph, and Kuzu. Next, we present key representative findings from our review.

DuckPGQ [32] is one of the few systems beyond Oracle PGX implementing SQL/PGQ path queries, the first version of one of the standard query languages for property graphs. It relies on a relational backend and deals with recursion either by unfolding it to several joins depending on the length of the paths or by resorting to multi-source BFS as an external module. However, the supported fragment of SQL/PGQ path queries is rather limited and it includes only the ANY SHORTEST WALK path mode. DuckPGQ implements a version of relational algebra enhanced with UDFs (user-defined functions) to support path queries evaluation.

MillenniumDB [34] supports all path modes of GQL with full RPQs, but only on the level of a single path. Combining sets of paths according to a formal algebraic framework goes beyond the current scope of MilleniumDB.

Neo4j [36] offers full support for finding trails and walks, but does not support arbitrary regular expressions to define all regular path queries. On the other hand, since the Cypher query language supports post-processing of the returned paths (viewed as lists) [12], non-recursive algebraic operations can be simulated in the system, but are not natively implemented.

Oracle/PGX [33] is a graph extension of the Oracle relational backend. It relies on Compressed Sparse Row storage and is capable of evaluating conjunctive RPQs under Walk, Trail, Simple and Acyclic semantics.

Kùzu [21] fully supports the walk semantics, but not for all regular expressions, and it limits the path length to 30. Trails and Acyclic walks are also supported. Similarly to Cypher, some post processing can be done on retrieved paths (which are stored as lists) in order to simulate non-recursive algebraic operations.

In conclusion, none of the current database systems incorporates a path-based algebraic query evaluation as presented in this work. Supporting such an algebra would facilitate the implementation of logical plans and the application of query optimization techniques in current and future graph database systems.

## 9 CONCLUSIONS AND FUTURE WORK

One of the key characteristics of graph queries is the ability to return paths instead of relations. Future versions of graph query languages, defined as part of the ISO/IEC standardization activities, will be able to manipulate paths and to ensure composability of queries. Our work is a considerable milestone towards this direction by providing a graph algebra that lays the foundations of composable graph queries. Such an algebra contains fundamental operators as in Codd's relational algebra but goes significantly beyond them by defining recursive operators and coverage of path modes (selectors and restrictors) defined in GQL and SQL/PGQ. The latter are defined by specifying grouping, order by and projection operators on path variables. Our work is accompanied by a parser implementing the algebraic operators.

Our work paves the way for further research on graph query algebraic optimizations leading to efficient implementations of graph query engines. While current graph databases including open-source and commercial ones are adding features from the standards (GQL and SQL/PGQ), they also need to provide internals for logical and physical plans inspired by a formal algebraic framework, as the one proposed in our paper.

## ACKNOWLEDGMENTS

This work was supported by ANID FONDECYT Chile through grant 1221727. R. García was supported by CONICYT-PFCHA / Doctorado Nacional / 2019-21192157. A. Bonifati was supported by ANR VeriGraph (nr. ANR-21-CE48-0015). D. Vrgoč was supported by FONDECYT Regular grant number 1240346.

## REFERENCES

[1] Renzo Angles. 2018. The Property Graph Database Model. In *Proc. Alberto Mendelzon International Workshop on Foundations of Data Management (AMW)*, Vol. 2100. CEUR Workshop Proceedings.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5 (2017), 68:1–68:40. https://doi.org/10.1145/3104031

[3] Diego Arroyuelo, Adrián Gómez-Brandón, Aidan Hogan, Gonzalo Navarro, and Javierl Rojas-Ledesma. 2024. Optimizing RPQs over a Compact Graph Representation. *The VLDB Journal* 33, 2 (2024), 349–374. https://doi.org/10.1007/s00778-023-00811-2

[4] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. Vol. 10. Morgan & Claypool Publishers. 1–184 pages. https://doi.org/10.2200/s00873ed1v01y201808dtm051

[5] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. 2003. Reasoning on regular path queries. *SIGMOD Rec.* 32, 4 (2003), 83–92. https://doi.org/10.1145/959060.959076

[6] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387. https://doi.org/10.1145/362384.362685

[7] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. 1987. A graphical query language supporting recursion. *SIGMOD Rec.* 16, 3 (dec 1987), 323–330. https://doi.org/10.1145/38714.38749

[8] Benjamín Farías, Carlos Rojas, and Domagoj Vrgoč. 2023. Evaluating Regular Path Queries in GQL and SQL/PGQ: How Far Can The Classical Algorithms Take Us? (2023). arXiv:2306.02194 http://arxiv.org/abs/2306.02194

[9] G. Fletcher, Jeroen Peters, and Alexandra Poulovassilis. 2016. Efficient regular path query evaluation using path indexes. In *International Conference on Extending Database Technology*. https://doi.org/10.5441/002/edbt.2016.67

[10] Nadime Francis, Amélie Gheerbrant, Paolo Guagliardo, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Liat Peterfreund, Alexandra Rogova, and Domagoj Vrgoč. 2023. A Researcher's Digest of GQL. In *Leibniz International Proceedings in Informatics (LIPIcs)*, Vol. 255. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 1:1–1:0. https://doi.org/10.4230/LIPIcs.ICDT.2023.1

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

[12] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the International Conference on Management of Data*

*(SIGMOD)*. ACM, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[13] Roberto García and Renzo Angles. 2022. An Algebra for Path Manipulation in Graph Databases. In *Advances in Databases and Information Systems*. Springer International Publishing, Cham, 61–74. https://doi.org/10.1007/978-3-031-15740-0_6

[14] Hector Garcia-Molina. 2008. *Database systems: the complete book*. Pearson Education India.

[15] M. Gondran. 1975. Path Algebra and Algorithms. In *Combinatorial Programming: Methods and Applications*. Springer Netherlands, Dordrecht, 137–148. https://doi.org/10.1007/978-94-011-7557-9_6

[16] X. Guo, H. Gao, and Z. Zou. 2021. Distributed processing of regular path queries in RDF graphs. *Knowl. Inf. Syst.* 63, 4 (2021), 993–1027. https://doi.org/10.1007/s10115-020-01536-2

[17] ISO. 2023. ISO/IEC 9075-16:2023 Information technology — Database languages SQL - Part 16: Property Graph Queries (SQL/PGQ). https://www.iso.org/standard/79473.html.

[18] ISO. 2024. ISO/IEC 39075:2024 Information technology — Database languages — GQL. https://www.iso.org/standard/76120.html.

[19] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaida. 2020. On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *International Conference on Management of Data (SIGMOD)*. ACM, New York, NY, USA, 681–697. https://doi.org/10.1145/3318464.3380567

[20] J. Kuijpers, G. Fletcher, T. Lindaaker, and N. Yakovets. 2021. Path indexing in the Cypher query pipeline. In *24th International Conference on Extending Database Technology (EDBT)*. Open Proceedings, 582–587. https://doi.org/10.1145/3299869.3319882

[21] Kuzu. 2024. Recursive relationship functions. https://docs.kuzudb.com/cypher/expressions/recursive-rel-functions/. [Accessed 06-10-2024].

[22] Leonid Libkin, Juan L. Reutter, Adrián Soto, and Domagoj Vrgoc. 2018. TriAL: A Navigational Algebra for RDF Triplestores. *ACM Trans. Database Syst.* 43, 1 (2018), 5:1–5:46. https://doi.org/10.1145/3154385

[23] Robert Manger. 2004. A new path algebra for finding paths in graphs. *26th International Conference on Information Technology Interfaces* (2004), 657–662 Vol.1.

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

[25] Alberto O. Mendelzon and Peter T. Wood. 1995. Finding Regular Simple Paths in Graph Databases. *SIAM J. Comput.* 24, 6 (1995), 1235–1258. https://doi.org/10.1137/S009753979122370X

[26] K. Miura, T. Amagasa, and H. Kitagawa. 2019. Accelerating regular path queries using FPGA. In *10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS)*. ACM, 47–54.

[27] Anil Pacaci, Angela Bonifati, and M. Tamer Özsu. 2022. Evaluating Complex Queries on Streaming Graphs. In *38th IEEE International Conference on Data Engineering (ICDE)*. 272–285. https://doi.org/10.1109/ICDE53745.2022.00025

[28] Juan L. Reutter, Adrián Soto, and Domagoj Vrgoc. 2021. Recursion in SPARQL. *Semantic Web* 12, 5 (2021), 711–740. https://doi.org/10.3233/SW-200401

[29] Marko A. Rodriguez and Peter Neubauer. 2011. A path algebra for multi-relational graphs. In *IEEE 27th International Conference on Data Engineering Workshops*. 128–131. https://doi.org/10.1109/ICDEW.2011.5767613

[30] 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. https://doi.org/10.1145/3434642

[31] Gábor Szárnyas, Jack Waudby, Benjamin A. Steer, Dávid Szakállas, Altan Birler, Mingxi Wu, Yuchen Zhang, and Peter A. Boncz. 2022. The LDBC Social Network Benchmark: Business Intelligence Workload. *Proc. VLDB Endow.* 16, 4 (2022), 877–890. https://doi.org/10.14778/3574245.3574270

[32] Daniel ten Wolde, Gábor Szárnyas, and Peter A. Boncz. 2023. DuckPGQ: Bringing SQL/PGQ to DuckDB. *Proc. VLDB Endow.* 16, 12 (2023), 4034–4037. https://doi.org/10.14778/3611540.3611614

[33] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the 4th International Workshop on Graph Data Management Experiences and Systems (GRADES)*. ACM, 7. https://doi.org/10.1145/2960414.2960421

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

[35] S. Wadhwa, A. Prasad, S. Ranu, A. Bagchi, and S. Bedathur. 2019. Efficiently answering regular simple path queries on large labeled networks. In *International Conference on Management of Data (SIGMOD)*. ACM, 1463–1480. https://doi.org/10.1145/3299869.3319882

[36] Jim Webber. 2012. A programmatic introduction to Neo4j. In *Proceedings of the ACM Conference on Systems, Programming, and Applications*. 217–218. https://doi.org/10.1145/2384716.2384777

[37] N. Yakovets, P. Godfrey, and J. Gryz. 2016. Query planning for evaluating SPARQL property paths. In *International Conference on Management of Data (SIGMOD)*. ACM, 1875–1889. https://doi.org/10.1145/2882903.2882944