# PGQL: a Property Graph Query Language

Oskar van Rest
Oracle Labs
oskar.van.rest@oracle.com

Sungpack Hong
Oracle Labs
sungpack.hong@oracle.com

Jinha Kim
Oracle Labs
jinha.kim@oracle.com

Xuming Meng
Oracle Labs
xuming.meng@oracle.com

Hassan Chafi
Oracle Labs
hassan.chafi@oracle.com

## ABSTRACT

**Graph-based approaches to data analysis have become more widespread, which has given need for a query language for graphs. Such a graph query language needs not only SQL-like functionality for querying structured data, but also intrinsic support for typical graph-style applications: reachability analysis, path finding and graph construction.**

**We propose a new query language for the popular Property Graph (PG) data model: the Property Graph Query Language (PGQL). PGQL is based on the paradigm of graph pattern matching, closely follows syntactic structures of SQL, and provides regular path queries with conditions on labels and properties to allow for reachability and path finding queries. Besides intrinsic *vertex*, *edge* and *path* types, PGQL also has the *graph* as intrinsic type and allows for graph construction and query composition.**

## CCS Concepts

**•Information systems → Query languages for non-relational engines;**

## Keywords

Graph query languages, graph-structured data, property graphs

## 1. INTRODUCTION

In recent years, both the data management community and the data mining community have been paying a lot of attention to the graph-based approaches in which graphs are used as fundamental representation for data analysis. The graph data model allows for promising analyses over topology, in addition to analyses over data stored in topology, which can be performed by standard relational mechanisms.

Essential for typical data analyses is a query language to query data in a high-level and productive manner. SQL, having been widely adopted as the query language for relational databases, illustrates very well the importance of query languages in data management. Now graph technology has become more widespread, the need has arisen for a well-designed query language for graphs that supports not only typical SQL-like queries over structured data, but also graph-style queries for reachability analysis, path finding, and graph construction. It is important that such a query language is general enough, but also has a right level between expressive power and ability to process queries efficiently on large-scale data.

Various graph query languages have been proposed in the past. Most notably, there is SPARQL, which is the standard query language for the RDF (Resource Description Framework) graph data model, and which has been adopted by several graph databases [1, 7, 12]. However, the RDF data model regularizes the graph representation as set of triples (or edges), which means that even constant literals are encoded as graph vertices. Such artificial vertices make it hard to express graph queries in a natural way. A more natural data model is the *Property Graph* (PG) data model, in which vertices and edges in a graph can be associated with arbitrary *properties* as key-value pairs. The PG data model has been adopted by various graph databases [5, 8, 11, 4] and graph processing frameworks [6, 2], and has formed the basis of recently proposed languages for graph algorithms [22, 10, 23] and graph queries [3, 21].

A notable query language for the PG data model is Cypher [3], which is, just like SPARQL, based on *graph pattern matching*, an elegant way of defining patterns in graphs. However, Cypher is missing some fundamental graph querying functionalities, namely, regular path queries and graph construction.

To overcome the limitations of existing graph query languages, we designed a new query language for the PG data model: the Property Graph Query Language (PGQL). PGQL combines graph pattern matching with SQL-like syntax and functionality and has full-blown support for regular path queries and graph construction. Because its syntax is SQL-like, the language is intuitive to use for existing SQL users. Furthermore, PGQL queries return a "result set" with variables and bindings, just like in SQL. This means that queries can be naturally nested inside SQL queries. In PGQL, a graph pattern is not only composed of vertices and edges, but also of paths, such that users can perform reachability analyses or find paths in a graph. Supported is the well-studied class of regular path queries, but not just with conditions on labels of edges, but with conditions on labels as well as properties of vertices and edges along paths. PGQL also has an intrinsic *graph* type to support graph transformation applications, allowing one or more graphs to be constructed and returned from a query.

Specifically, our contributions are as follows: *(i)* the design of PGQL, an intuitive high-level pattern-matching query language for the PG data model, which closely follows SQL's syntactic structures and provides SQL-like functionality as part of the solution (Section 3), *(ii)* the design of a syntax for regular path queries for PG graphs with conditions on labels and properties of vertices and


*Proceedings of the Fourth International Workshop on Graph Data Management Experience and Systems (GRADES 2016), June 24, 2016, Redwood Shores, USA*

edges along paths (Section 4), *(iii)* an elegant solution to graph construction inside a query language, allowing for graphs to be returned from a query and for graph queries to be composed (Section 5).

## 2. RELATED WORK

There are two types of languages for graphs. First, there are the *graph pattern-matching query languages* that are similar to PGQL. Most notably, SPARQL [9] is the standard query language for the RDF model. While SPARQL supports reachability by means of regular path queries (RPQs), actual paths that satisfy an RPQ cannot be returned from a query. Furthermore, conditions that compare vertices and edges along paths cannot be specified. Earlier proposals [17, 16, 19, 27], which were also based on a labeled graph, suffered from similar limitations. Cypher [3], the query language of the graph database Neo4j, was the first pattern-matching query language to target the PG model. While it has an intrinsic *path* type, path query support is based on finding all simple trails (i.e. paths with non-repeating edges). However, evaluation of such queries often requires iteration over the entire set of trails, which is an operation that is exponential in the size of the graph and therefore unsuitable for large graphs. Furthermore, Cypher does not have an intrinsic *graph* type to support graph construction applications. GraphQL [21] is a pattern-matching query language based on a PG-like model. It supports recursion in such a way that not only paths but even entire graph patterns can be the unit of repetition. However, the output of a GraphQL query is a set of graphs rather than a result set with variables and bindings. In practice we are often interested in the data (i.e. properties) inside vertices and edges and such a lanugage does not allow you to extract such information. PQL [24] is a pattern-matching query language for biological networks with SQL-like syntax. Its `WHERE` clause supports reachability patterns, while actual paths are constructed in the `SELECT` clause using pairs of bound variables from the `WHERE` clause.

Second, there are the more general purpose *imperative graph languages*. Green-Marl [22] is a language specifically designed to express graph analysis algorithms for the PG data model. The language has graph-specific data entities like *graph*, *vertex* and *edge* as intrinsic types and furthermore provides constructs for different graph traversals and iterations, such as Breadth-First Search (BFS), Depth-First Search (DFS), incoming neighbor iteration, outgoing neighbor iteration, etc. GRAPHiQL [23] is a graph language for relational databases, based on Pregel-style vertex-centric programming. The language introduces the Graph Tables data model, which is a representation of a graph that can be compiled easily into a set of relational tables. Gremlin [10] is a graph traversal language for the PG data model. It uses JVM-based languages such as Java and Groovy as host language. While Green-Marl, Gremlin and GRAPHiQL can be used to implement graph queries, their imperative nature makes them more suitable for expressing graph analysis algorithms such as PageRank, Betweenness Centrality, etc.

Apart from languages specifically designed for graph querying, there are a number of existing query languages being considered for reuse, possibly by means of language extension. These languages include: SQL, the query language for relational databases, Datalog, a declarative logic programming language that has natural support for recursion, XPath, a query language for XML data, and, OCL, a constraint and query languages for the UML model. However, the problem with repurposing such languages for graph querying is the mismatch between data models, which burdens the user by requiring them to translate between models.

```
1  SELECT   friend.name, friend.age
2  //       'Andy'       12 (query solutions in comment)
3  FROM snGraph
4  WHERE
5    (x WITH name = 'Paul') -[:likes]-> (friend),
6    (y WITH name = 'Amber') -/:likes*1..2/-> (friend),
7    x.age >= 2 * friend.age
8  ORDER BY friend.name
```

**Figure 1:** Example PGQL query, returning the friends of Paul that are also friends, or friends of friends, of Amber. Only friends that are at least twice as young are returned.

## 3. PATTERN MATCHING QUERIES

PGQL is based on the paradigm of *graph pattern matching*, which is an elegant way of defining pattern in graphs (Section 3.1). It combines this with SQL-like syntax and also provides SQL-like functionality as part of the solution (Section 3.2).

### 3.1 Graph Pattern Matching

A PGQL query is composed of the three clauses `SELECT`, `FROM` and `WHERE`, which are followed by optional solution modifier clauses such as `ORDER BY`, `GROUP BY`, and `LIMIT`. The `FROM` clause can also be omitted when there is only one graph instance. Additionally, PGQL includes special operators for graph pattern matching: vertex matching, edge matching and path matching. These matching operators are placed in the `WHERE` clause along with predicates over vertices and edges.

Figure 1 shows an example PGQL query which finds patterns from a data graph named `snGraph` (line 3). In the `WHERE` clause, the query matches a vertex `x` that has a property `name` with value `'Paul'`. The vertex `x` has an edge with the label `'likes'`. The destination vertex of this edge is referred to as vertex `friend` (line 5). Similarly, the query matches another vertex `y` with its `name` being `'Amber'`. The vertex `y` is also connected to the vertex `friend` but through a path; the path is composed only of edges with label `'likes'` and the (hop-)length of the path is between `1` and `2` inclusively (line 6). Line 7 dictates that the value of property `age` in vertex `x` is at least twice as large to that of vertex `friend`. All the instances that match with this pattern are first sorted by `name` values of `friend` vertices (line 7), before the `name` and `age` property values of the `friend` vertex are returned (line 1).

Just like in SQL, the result of a PGQL query forms a tabular "result set" with variables and their bindings. This allows PGQL queries to be naturally nested inside SQL queries. PGQL also has intrinsic data types for graph-specific entities: *vertex*, *edge*, *path* and *graph* – the result set bindings can be any of these graph-specific types. Graph-specific entities can be returned or operated on, for example, using built-in functions such as `id()` for accessing the identifier of a vertex or edges, `label()` for accessing the label of an edge, `labels()` for accessing the labels of a vertex, and `length()` and `weight()` for accessing length and weights of paths.

In the PG data model, a graph has vertices and directed edges that have unique identifiers and properties, which are arbitrary key-value pairs. Properties may also be sets, but this is not further discussed in the paper. Edges have a single label, while vertices have a set of labels that are typically used to encode types when there are multiple classes of vertices. Figure 2 shows an example PG graph named `snGraph`. This graph will function as the input graph to most of the example queries in this paper. In the graph, the vertex with identifier `100` has the label `Person`, a property `name` with the value `'Amber'`, a property `age` with the value `29`, and a property `eyeColor` with the value `'Brown'` The edge with identifier `0` going from vertex `100` to vertex `400` has the label `likes` and a property `since` with the value `'2016-04-03'`. If we evaluate the example

query from Figure 1 on top of the graph, we obtain a single result that has `friend.name` bound to `'Andy'` and `friend.age` bound to value `12`. Note that for example queries in the paper, we place query solutions inside code comments (see line 2 in Figure 1).

PGQL supports two different pattern matching semantics: isomorphism and homomorphism. Under isomorphic semantic, two different *query* vertices or edges are not allowed to map to the same *data* vertex or edge. There is no such a restriction under homomorphic semantic. In PGQL, the default semantic is isomorphism, but homomorphism can be used by specifying `WHERE ALL` instead of `WHERE`. For example, in Figure 3, we query a graph that represents a code base with functions and function calls. The query finds all functions that are called by the function `'init_pci'`. Since we use homomorphic semantic (i.e. `WHERE ALL`), the recursive function call for which both the vertices `f1` and `f2` bind to the same data vertex `10`, also matches the pattern and thus becomes an output solution. However, the recursive function call does not match, had we chosen for isomorphism, since vertices `f1` and `f2` would not be allowed to bind to the same data vertex. With `WHERE ALL`, it is also possible to specify more finegrained mappings between query pattern and data graph, by specifying nonequality constraints such as `f1 != f2`.

Finally, it is possible to specify two or more *disconnected* graph patterns inside the `WHERE` clause. This results in a cartesian product of the matches for the different patterns.

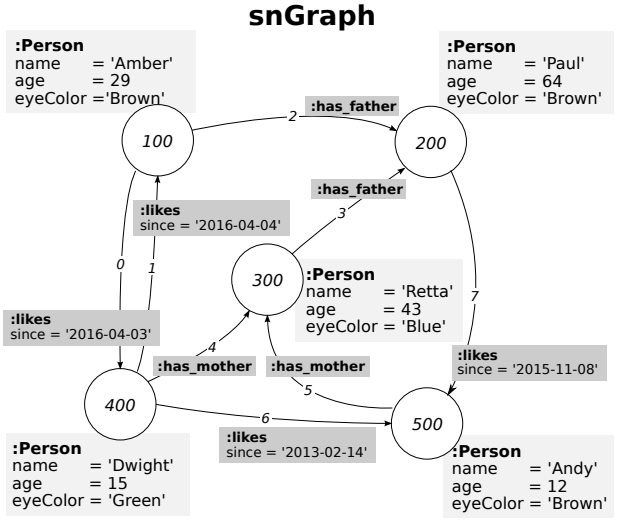


**Figure 2:** Example Property Graph. Here, `100-500` are vertex identifiers, `Person` is a vertex label, `name`, `age`, and `eyeColor` are vertex properties, `0-7` are edge identifiers, `likes`, `has_father` and `has_mother` are edge labels, and `since` is an edge property.

```
1  SELECT    f2.name
2  //        'init_pci'
3  //        'dep_pgb'
4  FROM CodeBaseGraph
5  WHERE ALL
6    f1 -[e:calls]-> f2,
7    f1.name = 'init_pci'
```

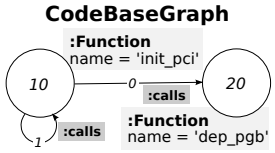


**Figure 3:** A query that uses subgraph *homomorphic* semantic rather than PGQL's default subgraph *isomorphic* semantic. In the query, we find all functions that are called by function `'init_pci'`. Because we use homomorphic semantic (`WHERE ALL` instead of `WHERE`), the recursive functional call is also a solution, even though it binds both `f1` and `f2` bind to the same vertex (i.e. vertex *10*).

```
1  SELECT     person.eyeColor, COUNT(*), MAX(person.age)
2  //         'Brown'          1         29
3  //         'Blue'           1         43
4  FROM snGraph
5  WHERE
6    (person:Person),
7    NOT EXISTS {
8      person -[:has_father|:has_mother]-> parent,
9      parent.eyeColor = 'Blue'
10   }
11 GROUP BY person.eyeColor
12 HAVING COUNT(*) < 4
13 ORDER BY COUNT(*) DESC
```

**Figure 4:** Overview of SQL-like constructs in PGQL: `COUNT`, `MAX`, and `AVG` are aggregates, `NOT EXISTS` tests for nonexistence of a graph pattern (i.e. `person` does not have a child with green eyes); `GROUP BY` groups pattern matches based on outcome values for one or more expressions (i.e. make groups such that persons in a group have the same `eyeColor`); `HAVING` filters whole groups from the result (i.e. only keep groups with a size smaller than `4`); `ORDER BY` orders solutions (i.e. order by group size in descending manner).

## 3.2 SQL-Like Functionality

On top of graph pattern matching, PGQL provides SQL-like querying functionality. It also closely follows SQL's syntactic structures.

### *3.2.1 Grouping and Aggregation*

Pattern matches can be grouped using `GROUP BY`, which is typically used in combination with aggregates like `MIN`, `MAX`, `AVG`, `SUM` to aggregate over the solutions in a group. The `HAVING` construct can be used in addition to `GROUP BY`, to filter out whole groups from the solution. For example, in Figure 4, the pattern matches are grouped by a vertex property `person.eyeColor` (line 11). Then, groups are filtered out that have more than three elements (line 12).

### *3.2.2 Order By, Offset, Limit*

Solutions can be sorted using `ORDER BY` (see Figure 4, line 13). Like in SQL, the construct takes one or more order expressions that each can have an optional order modifier (`ASC` or `DESC`). Furthermore, `OFFSET` and `LIMIT` can be used to select different subsets from the query solutions, for example, only results 5 to 10. Note that normally, this is only meaningful if the order is made predictable by means of `ORDER BY`.

### *3.2.3 Union*

Using the `UNION` construct, one can match alternative patterns and combine their solutions. Although we do not provide an example here, the functionality is similar to the `UNION` in SQL. `UNION` *removes* duplicate rows, while `UNION ALL` *maintains* duplicate rows. Besides the `UNION` for alternative matching, PGQL also introduces a `UNION` *aggregate* (Section 5), which is used inside the `SELECT` to take the union of a set of graphs in order to construct a larger graph.

### *3.2.4 Negation*

The `NOT EXISTS` construct tests for the nonexistence of a graph pattern, given a set of bound variables from the outer scope. This is similar to the `NOT EXISTS` in SPARQL. For example, in Figure 4, we filter out pattern matches that have a binding for the vertex `person` that has an incoming `has_father` or `has_mother` edge from a vertex `child` that has a property `eyeColor` with value `Green`. The `NOT EXISTS`, as well as the non-negating form `EXISTS`, are part of the expression system and can be used anywhere where expressions are allowed, for example in the `SELECT`, `ORDER BY` and `GROUP BY` clauses.

### 3.2.5 Subqueries

Subqueries provide a means to compose queries and allow users to specify more complex search patterns. Subqueries in PGQL can be specified in the `FROM` and `WHERE` clauses and they are processed in a bottom-up fashion. Examples of subqueries are provided in Section 5.3. PGQL does not have a concept of correlated subqueries like in SQL, but *EXISTS* and *NOT EXISTS* (see above) provide similar but more limited functionality. In a future version of PGQL, we may support correlated subqueries to overcome this limitation. This has also been suggested [14] for SPARQL.

# 4. PATH QUERIES

Path queries, a fundamental graph query paradigm, allow for testing of the existance of a path between pairs of vertices and also for obtaining such paths to return them from queries. This inherently recursive functionality reveals the power of graph technology: the traditional relational database has only weak support for recursion. The applications are too numerous to describe here, but examples include static code analysis [20], fraud detection, network routing and road navigation.

A well-studied class of path query is the so-called *Regular Path Query* (RPQ) [27, 15, 26, 25], which finds vertices connected by a path such that the labels of the edges along the path satisfy a certain regular expression. RPQs form an integral part of many graph query languages [17, 16, 19, 27], including SPARQL [9], the standard RDF graph query language. However, while such regular expressions over edge labels make sense for labelled graphs, the functionality is too limited for PG graphs that store information needed for certain analysis inside vertex and edge properties rather than labels. Therefore, PGQL extends the class of RPQs with general expressions over vertices and edges along paths. Furthermore, PGQL allows for comparing vertices and edges along paths while still maintaining the same query evaluation complexity as usual RPQs. Reachability queries are discussed in Section 4.1, while path finding queries are discussed in Section 4.2.

## 4.1 Reachability Queries

Reachability queries test for the existence of paths between pairs of vertices. In PGQL, such queries take the form of a graph pattern that not only consists of vertices and edge, but is also composed of one or more regular paths. Such a regular path is a pattern that is declared at the beginning of the query by means of the `PATH` construct. Once it is declared, a path pattern is embedded in the graph pattern by referring the path pattern from the `WHERE` clause. Not only graph patterns are composed by path patterns in this way, but path patterns themselves can also be composed of other path pattern. This allows for expressing more complex regular path queries.

Figure 5 is an example reachability query. It describes a path pattern `has_new_friend` that has a source vertex `s` and an outgoing edge to a destination vertex `t` with the label `Person`. The edge has the label `likes` and also a property `since` with a value greater than or equal to `2016-01-01` (lines 1-2). In the `WHERE` clause, the path query connectors `-/` and `/->` describe that vertices `x` and `y` are connected via a regular path pattern, rather than an edge. Here, they are connected by `1` to `2` applications of the `has_new_friend` pattern (line 7). In the first application, vertex `s` will bind to the data vertex that vertex `x` is bound to, but in a follow-up application, `s` will bind to the data vertex that `t` was bound to in the previous application. Figure 6 shows another reachability query with a path pattern `has_parent` that has a `WHERE` clause with additional predicates (lines 3-7). The first predicate (line 3) dictates that at least one of the vertices `c` or `p` has a property `eyeColor` with the value `'Brown'`, while the second predicate (line 4-7) dictates that the vertex `p` does not have an incoming `has_father` or `has_mother` edge from a vertex `c2` that has a property `eyeColor` with the value `'Brown'`. A second path pattern is composed of the `has_parent` pattern by unbounded repetition, such that a `has_ancestor` relation is described (line 8). Then, in the `WHERE` clause (line 14), the graph pattern is composed of the `has_ancestor` path pattern. Figure 7 presents an overview of the various path pattern constructs.

```
1  PATH has_new_friend :=
2    (s) -[:likes WITH since >= '2016-01-01']-> (t:Person)
3  SELECT   y.name
4  //       'Dwight'
5  FROM snGraph
6  WHERE
7    (x:Person) -/has_new_friend*1..2/-> (y),
8    x.name = 'Amber'
```

**Figure 5:** An example reachability query that find friends, or friends of friends, of Amber, by only following `likes` edges created on or after `2016-01-01`.

```
1  PATH has_parent :=
2    (c:Person) -[:has_father|:has_mother]-> (p:Person)
3    WHERE c.eyeColor = 'Brown' OR p.eyeColor = 'Brown',
4          NOT EXISTS {
5            (c2:Person) -[:has_father|:has_mother]-> p,
6            c2.eyeColor = 'Brown'
7          }
8  PATH has_ancestor := () -/has_parent*/-> ()
9  SELECT   y.name,  ancestor.name
10 //       'Dwight' 'Paul'
11 FROM snGraph
12 WHERE
13   (x:Person) -[:likes]-> (y:Person), x.name = 'Amber',
14   x -/has_ancestor/-> ancestor <-/has_ancestor/- y
```

**Figure 6:** Find the persons that Amber likes and who have a common ancestor. For each child and their parent on a path to an ancestor, it holds that either the child or the parent has brown eyes (line 3), and, that the parent does not have another child with brown eyes (lines 4-7).

```
1  PATH forwardEdge      := () -[e]-> ()
2  PATH reverseEdge      := () <-[e]- ()
3  PATH sequencePath     := () -[e1]-> () -[e2]-> ()
4  PATH compositePath    := () -/sequencePath/-> ()
5  PATH labelRepetition  := () -/:lbl*1..4/-> ()
6  PATH alternativeLabel := () -/:lbl1|:lbl2/-> ()
7  PATH pathRepetition   := () -/path*1..4/-> ()
8  PATH alternativePath  := () -/path1|path2/-> ()
9  SELECT x, y, p1, p2, $kShortest(myGraph, x, y, 20) AS p3
10 FROM myGraph
11 WHERE
12   x -/sequencePath*/-> y,         // reachability
13   x -/p1:lbl*/-> y,               // min-hop shortest
14   x -/p2^'weight_prop':lbl*/-> y // weighted shortest
```

**Figure 7:** An overview of path query syntax in PGQL. Note: this query does not implement a meaningful analysis.

## 4.2 Path Finding Queries

Path finding queries are queries that find paths between pairs of vertices such that they can be returned, compared, or operated on. Different use cases demand for different kinds of paths to be found. Sometimes, we only want to find a *single* shortest path (per source-destination pair) with a minimal hop distance or a minimal weight. For other use cases it migh be required to obtain *multiple* paths, for example, k shortest paths [18] or k dissimilar paths [13]. Although these different path finding semantics are all very useful, a query language can't have all of them "built-in" without significantly increasing the complexity of the language. Therefore, because single shortest path finding is the most common case, it is choses as

the built-in path finding semantic for PGQL. However, path finding algorithms with different semantics can be used with PGQL by means of user-defined functions that implement such algorithms. Such functions take a graph, a source vertex, a destination vertex and other parameters as input and return a path or a set of paths as output. These functions could be provided by the graph database or framework, or they may be implemented by users that need very specific path finding semantics.

The overview in Figure 7 includes the path finding functionality (lines 9, 13-14). Path `p1` binds to a single minimal hop shortest path (per source-destination pair). Path `p2` binds to a single minimal weight shortest path. The weight is computed by summing over the nonnegative values for property `'weight_prop'` of the edges along the path. In the less common case that a specialized weight function is required — for example one that includes two edges properties rather than one — it is possible to construct a graph (see Section 5) with a new edge property that holds the result of the weight function. Finally, path `p3` binds to the `20` shortest paths, provided that `kShortest` is a user-defined function that implements *k* shortest path finding.

# 5. GRAPH CONSTRUCTION

Graph construction queries extract entities from an input graph and possibly enhance them with additional vertices, edges, or properties, in order to produce one or more output graphs. Such functionality is essential for query compositionality, such that complex queries can be decomposed into simpler ones, queries can be executed in sequence, and, user-specific *views* — which is a typical database functionality — can be implemented.

In SPARQL, graph construction is supported by means of the `CONSTRUCT` query form. This form has not only a graph pattern for matching a subgraph, but also a pattern for producing a new graph. While the `SELECT` query returns a tabular "result set", the `CONSTRUCT` query returns the output graph as a special kind of result. However, the limitation of the `CONSTRUCT` query is that it only allows for returning a single graph that is the union of all the subgraphs obtained by the query. SPARQL does not allow for returning individual subgraphs or for using `GROUP BY` in combination with `CONSTRUCT` to allow subgraphs to be grouped such that a graph can be returned for each of the groups. To overcome these limitiations, PGQL allows graph construction patterns to be specified in the `SELECT` clause such that multiple graphs can be constructed and returned as part of the result set, unifying the way in which data is returned from queries: as a result set (Section 5.1). Using a new aggregate `UNION`, PGQL allows groups of solutions to be aggregated into one or more larger graphs (Section 5.2). Furthermore, queries can be composed by allowing a query to extract a graph from a result set that is returned by another query (Section 5.3).

## 5.1 Graph Construction in SELECT

New graphs can be constructed by specifying one or more graph construction patterns inside the `SELECT` clause. These patterns consist of entities from the `WHERE` clause, but may also introduce new vertices, edges and properties. The graphs patterns are optionally assigned a variable name and are returned as part of the result set.

Figure 8 shows an example in which we first match all the child-father relations in the `snGraph` (line 7) and then construct a graph (lines 1-5) that consists of the original `child` vertex, `has_father` edge and `father` vertex (line 2), a new `has_child` edge from father to child (line 3), and a new property `gender` for the vertex `father` with the value set to `'MALE'` (line 4). The graph is assigned a variable name `childFatherGraph`. Since the pattern is matched twice, the result set contains two rows, each holding one graph.

```
1  SELECT GRAPH{
2    child -[e1]-> father,
3    father -[:has_child]-> child,
4    father.gender := 'MALE'
5  } AS childFatherGraph
6  FROM snGraph
7  WHERE (child) -[e1:has_father]-> (father)
```

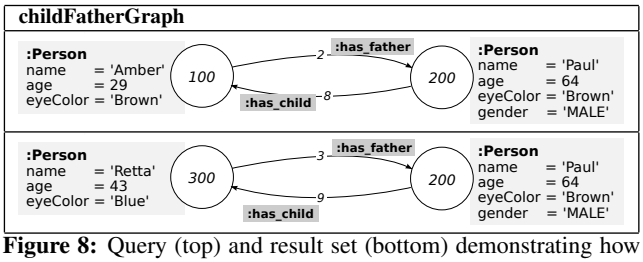


**Figure 8:** Query (top) and result set (bottom) demonstrating how graphs are constructed by specifying a graph construction pattern in the `SELECT` and how graphs are returned as part of the result set.

```
1  SELECT UNION(GRAPH{x -[e]-> y}) AS parentChildrenGraph
2  FROM snGraph
3  WHERE
4    x -[e:has_father|:has_mother]-> y
5  GROUP BY e.label()
```

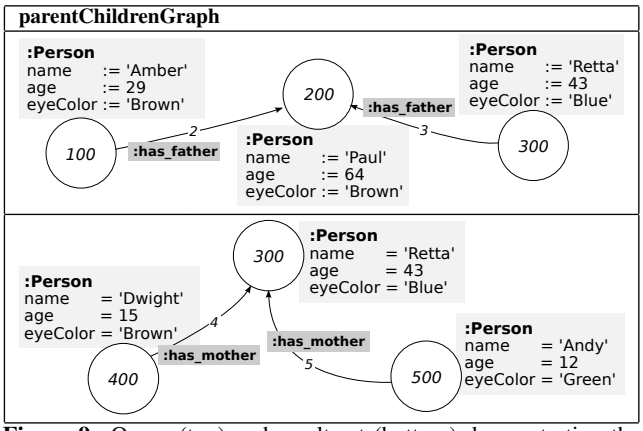


**Figure 9:** Query (top) and result set (bottom) demonstrating the *UNION* aggregate, which takes a set of graphs as input and returns a single graph as output.

## 5.2 Union of Graphs

We introduce a new `UNION` aggregate that takes a set of graphs as input and produces a single output graph. Vertices and edges that occur repeatedly in different input graphs are mapped into the same vertex or edge in the output graph. Just like other aggregates such as `MIN`, `MAX` and `AVG`, `UNION` aggregates over a group of solutions, such that when a grouping is specified using `GROUP BY`, multiple graphs may be returned. For example, in Figure 9, we first find all `has_father` and `has_mother` relations (line 4) and then group the relations by their type (line 5) and construct a graph for each group using the `UNION` aggregate (line 1).

```
1  SELECT   p.name, p.age
2  //       'Amber' 29
3  FROM likesGraph IN {
4    SELECT UNION(GRAPH{x -[e]-> y}) AS likesGraph
5    FROM snGraph
6    WHERE (x) -[e:likes]-> (y),
7          e.since > '2016-01-01'
8  }
9  WHERE
10   (p:Person), p.age = maxAge,
11   { SELECT MAX(v.age) AS maxAge WHERE (v) }
```

**Figure 10:** Query composition is supported by means of subqueries in the `FROM` and `WHERE` clauses.

## 5.3 Query Composition

Query composition in PGQL is just like in SQL based on subqueries. Subqueries in PGQL can be specified in the `WHERE` and `FROM` clauses and they are processed in a bottom-up fashion: innermost subqueries provide bindings to outer queries. Figure 10 shows an example query with a subquery in the `WHERE` clause. The subquery matches all vertices and returns the maximal value for the property `age` of the vertices in the graph (line 11). In the outer query, we match all the persons in the graph that have the maximum age (line 10). Subqueries may also return vertices and edges that can become part of the outer query's graph pattern.

A subquery in the `FROM` clause, however, has to return at least one graph that can be used as input to the outer query. For example, in Figure 10, we first match all the `likes` edges that were created after `'20016-01-01'` (lines 6-7) and combine them into a graph named `likesGraph` (line 4). Then, we extract the `likesGraph` from the result set to make it the input graph of the outer query (line 3). With the graph `snGraph` as input to the subquery, the outer query returns a single result in which `p.name` is bound to `'Amber'` and `p.age` is bound to `29`.

# 6. CONCLUSION

We introduced PGQL, an intuitive SQL-like pattern-matching query language for the Property Graph (PG) data model. We believe that PGQL has the essential graph query language ingredients, namely, SQL-like functionality, graph entities (i.e. *vertex*, *edge*, *path*, *graph*) as intrinsic types, and support for regular path queries, path finding and graph construction. PGQL's SQL-like syntax provides familiarity for existing SQL users and its tabular "result set" provide a uniform interface for accessing returned entities, including vertices, edges, paths and graphs. The tabular output also allows PGQL queries to be naturally nested inside SQL queries such that the language can be easily integrated into existing database technology.

# 7. ACKNOWLEDGMENTS

We thank our team members from Oracle Labs Parallel Graph Analytics (PGX) as well as the teams behind Oracle Big Data Spatial and Graph, Oracle Labs Frappé, Sparsity Technologies and LDBC's Graph Query Language Task Work Force, for their valuable ideas, insights and contributions.

# 8. REFERENCES

[1] AllegroGraph. http://franz.com/agraph/allegrograph/.

[2] Apache TinkerPop. http://tinkerpop.incubator.apache.org.

[3] Cypher - the Neo4j query Language. http://www.neo4j.org/learn/cypher.

[4] InfiniteGraph. http://www.objectivity.com/infinitegraph.

[5] Neo4j graph database. http://www.neo4j.org/.

[6] Oracle Parallel Graph Analytics (PGX). http://www.oracle.com/technetwork/oracle-labs/parallel-graph-analytics.

[7] Oracle Spatial and Graph, RDF Semantic Graph,. http://www.oracle.com/technetwork/database/options/spatialandgraph/overview/rdfsemantic-graph-1902016.html.

[8] Sparksee by Sparsity Technologies. http://sparsity-technologies.com/.

[9] SPARQL Query Language for RDF. http://www.w3.org/TR/rdf-sparql-query/.

[10] Tinkerpop, Gremlin. https://github.com/tinkerpop/gremlin/wiki.

[11] Titan Distributed Graph Database. http://thinkaurelius.github.io/titan/.

[12] Virtuoso Universal Server. http://virtuoso.openlinksw.com/.

[13] V. Akgün, E. Erkut, and R. Batta. On finding dissimilar paths. *European Journal of Operational Research*, 121(2):232–246, 2000.

[14] R. Angles and C. Gutierrez. Subqueries in sparql. In *AMW*. Citeseer, 2011.

[15] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *ACM Transactions on Database Systems (TODS)*, 37(4):31, 2012.

[16] M. P. Consens and A. O. Mendelzon. Expressing structural hypertext queries in graphlog. In *Proceedings of the second annual ACM conference on Hypertext*, pages 269–292. ACM, 1989.

[17] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *ACM SIGMOD Record*, volume 16, pages 323–330. ACM, 1987.

[18] D. Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2):652–673, 1998.

[19] R. H. Güting. Graphdb: Modeling and querying graphs in databases. In *VLDB*, volume 94, pages 12–15. Citeseer, 1994.

[20] N. Hawes, B. Barham, and C. Cifuentes. Frappé: Querying the linux kernel dependency graph. In *Proceedings of the GRADES'15*, page 4. ACM, 2015.

[21] H. He and A. K. Singh. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 405–418. ACM, 2008.

[22] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for Easy and Efficient Graph Analysis. In *ASPLOS*, pages 349–362. ACM, 2012.

[23] A. Jindal and S. Madden. Graphiql: A graph intuitive query language for relational databases. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 441–450. IEEE, 2014.

[24] U. Leser. A query language for biological networks. *Bioinformatics*, 21(suppl 2):ii33–ii39, 2005.

[25] L. Libkin, T. Tan, and D. Vrgoč. Regular expressions for data words. *Journal of Computer and System Sciences*, 81(7):1278–1297, 2015.

[26] L. Libkin and D. Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, pages 74–85. ACM, 2012.

[27] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM Journal on Computing*, 24(6):1235–1258, 1995.