PGQL 0.9 Specification

Table of Contents

Introduction

PGQL (Property Graph Query Language) is a query language for the Property Graph (PG) data model. This specification defines the syntax and semantics of PGQL.

Essentially, PGQL is a graph pattern-matching query language. A PGQL query describes a graph pattern with nodes, edges, properties, and their relationships, When the query is evaluated against a Property Graph instance, the query engine finds all subgraph instances of the graph that match to the specified query pattern. Then the query engine returns the selected data entities from each of the matched subgraph instance.

Example:

Consider the following example PGQL query:

Example PGQL query

```
SELECT m.name, o.name
WHERE (n WITH type = 'Person' AND name = 'John') -[e1 WITH type = 'friendOf']-> (m
WITH type = 'Person') <-[e2 WITH type = 'belongs_to']- (o WITH type = 'Car')
```

In the WHERE clause, the above query defines the pattern to be found.

- The pattern is composed of three nodes (*n*, *m*, and *o)* and two edges (*e1* and *e2)*.
- There is an edge (*e1*) from node *n* to node *m*.
- There is an edge (*e2*) from node *o* to node *m*.
- Nodes *n* and *m* have a property *type* with value *'Person'*, while node *o* has a property *type* with value *'Car'*
- Node *n* has another property *name* with value *'John'*.
- Edges *e1* and *e2* both have a property *type* whose values are *'friendOf'* and *'belongs_to'* respectively.

In the SELECT clause, the above query defines the data entities to be returned.

- For each of the matched subgraph, the query returns the property *name* of node *m* and the property *name* of node *o*.

Basic Query Structure

The syntax structure of PGQL resembles that of SQL (Standard Query Language) of relational database systems. A basic PGQL query consists of the following three clauses:

```
<BASIC PGQL QUERY> :=
    <SELECT clause>
    <WHERE clause>
    (<SOLUTION MODIFIER clause>)
```

- The SELECT clause defines the data entities that are returned in the result.
- The WHERE clause defines the graph pattern that is matched against the data graph instance.
- The SOLUTION MODIFIER clause defines additional operations for building up the result of the query. The SOLUTION MODIFIER clause is optional.

The detailed syntax and semantic of each clause are explained in following sections.

WHERE Clause

In a PGQL query, the WHERE clause defines the graph pattern to be matched.

Syntactically, a WHERE clause is composed of the keyword WHERE followed by a comma-separated sequence of constraints.

```
<WHERE clause> := <Constraint> (, <Constraint>)*
<Constraint> := <Topology Constraint>
              | <Value Constraint>
              | <In-lined Constraint>
```

Each constraint is one of the following types:

- A *topology* constraint describes a partial topology of the subgraph pattern, i.e. nodes and edges in the pattern.

- A *value* constraint describes a general constraint other than the topology; the constraint takes the form of a Boolean expression which typically involves property values of the nodes and edges.
- An *in-lined* constraint is a syntactic sugar where value constraints are written inside node terms or edge terms of a topology constraint.

There can be multiple constraints in the WHERE clause of a PGQL query. Semantically, all constraints are conjunctive – that is, each matched result should satisfy every constraint in the WHERE clause.

During query evaluation, the nodes and edges in the query pattern are bound to nodes and edges in a data graph. Whether multiple nodes in the pattern may bind to the same node in the data graph depends on the chosen pattern matching semantic. Two popular pattern matching semantics are described here.

Topology Constraint

A topology constraint describes a partial topology of the subgraph pattern. In other words, a topology constraint describes some connectivity relationships between nodes and edges in the pattern, whereas the whole topology of the pattern is described with one or multiple topology constraints.

A topology constraint is composed of one or more node terms and edge terms. As the name specifies, a node term corresponds to a node in the pattern, an edge term to an edge. In a query, each node term or edge term is (optionally) associated with a *variable*, which is a symbolic name to refer the corresponding node or edge in the pattern. For example, consider the following topology constraint:

```
(n)-[e]->(m)
```

The above example defines two nodes (with variable names *n* and *m*), and an edge (with variable name *e*) between them. Also the edge is directed such that the edge e is an outgoing edge from node n.

More specifically, a node term is written as a variable name inside a pair of parenthesis *()*. An edge term is written as a variable name inside a square bracket *[]* with two dashes and an inequality symbol attached to it – which makes it look like an arrow drawn in ASCII art. An edge term is always connected with two node terms as for the source and destination node of the edge; the source node is located at the tail of the ASCII arrow and the destination at the head of the ASCII arrow.

Repeated Variables in Multiple Topology Constraints

There can be multiple topology constraints in the WHERE clause of a PGQL query. In such a case, node terms that have the same variable name correspond to the same node entity. For example, consider the following two lines of topology constraints:

```
(n)-[e1]->(m1),
  (n)-[e2]->(m2)
```

Here, the node term (n) in the first constraint indeed refers to the same node as the node term (n) in the second constraint. It is an error, however, if two edge terms have the same variable name, or, if the same variable name is assigned to an edge term as well as to a node term in a single query.

Syntactic Sugars for Topology Constraints

For user's convenience, PGQL provides several syntactic sugars (short-cuts) for topology constraints.

First, a single topology constraint can be written as a chain of edge terms such that two consecutive edge terms share the common node term in between. For instance, the following topology constraint is valid in PGQL:

```
(n1)-[e1]->(n2)-[e2]->(n3)-[e3]->(n4)
```

In fact, the above constraint is equivalent to the following set of comma-separated constraints:

```
(n1)-[e1]->(n2),
  (n2)-[e2]->(n3),
  (n3)-[e3]->(n4)
```

Second, PGQL syntax allows to reverse the direction of an edge in the query, i.e. right-to-left instead of left-to-right. Therefore, the following is a valid topology constraint in PGQL:

```
(n1)-[e1]->(n2)<-[e2]-(n3)
```

Please mind the edge directions in the above query -- node n2 is a common outgoing neighbor of both node n1 and node n3.

Third, PGQL allows to omit *not-interesting* variable names in the query. A variable name is not interesting if that name would not appear in any other constraint, nor in other clauses (SELECT, SOLUTION MODIFIER). As for a node term, only the variable name is omitted, resulting in an empty parenthesis pair. In case of an edge term, the whole square bracket is omitted in addition to the variable name. In this case, the remaining ASCII arrow can have either one dash or two dashes.

The following table summarizes these short cuts.

Syntax Sugars for Omitting Variable Names	
Basic form	`(n)-[e]->(m)`
Omit variable name of the source node	`()-[e]->(m)`
Omit variable name of the destination node	`(n)-[e]->()`
Omit variable names in both nodes	`()-[e]->()`
Omit variable name in edge	`(n)-->(m)`
Omit variable name in edge (alternative, one dash)	`(n)->(m)`
Omitting variables in both node and edge	`k1->()->()->k2`

Finally, the parenthesis in the node term can be omitted, if the node term is attached to an edge term and there is no in-lined value constraints. Therefore the following syntax is a valid topology constraint in PGQL.

```
x->y->z<-w-[e1]->q
```

Disconnected Topology Constraints

In the case the topology constraints form multiple groups of nodes and edges that a not connected to each other, the semantic is that the different groups are matched independently and that final result is produced by taking the Cartesian product of the result sets of the different groups. The following is an example of a query that will result in a Cartesian product.

```
SELECT *
WHERE
  n1 -> m1
  n2 -> m2 // nodes {n2, m2} are not connected to nodes {n1, m1}, resulting in a
Cartesian product
```

Value Constraint

The value constraint describes a general constraint other than the topology. A value constraint takes the form of a Boolean expression which typically involves certain *property values* of the nodes and edges that are defined in topology constraints in the same query. For instance, the following example consists of three constraints – one topology constraint followed by two value constraints.

```
x -> y,
x.name = 'John',
y.age > 25
```

In the above example, the first value constraint demands that the node x has a property *name* and its value to be '*John*'. Similarly, the second value constrain demands that the node y has a numeric property *age* and its value to be larger than 25. Here, in the value constraint expressions, the dot (.) operator is used for property access. For the detailed syntax and semantic of expressions, please refer to the corresponding section in this document.

Note that in PGQL the ordering of constraints does not has any effect on the result. Therefore, the previous example is equivalent to the following:

```
x.name = 'John',
x -> y,
y.age > 25
```

In-lined Constraint

An *in-lined* constraint is a syntactic sugar where value constraints are written directly inside a topology constraint. More specifically, expressions that access the property values of a certain node (or edge) are put directly inside the parenthesis (or the square bracket) of the corresponding node (or edge) term. Consider the following set of constraints.

```
n-[e]->(),
n.name = 'John' OR n.name = 'James',
n.type = 'Person'
e.type = 'workAt',
e.workHours < 40
```

The above constraints can re-written with in-lined constraint as follows:

```
(n WITH name = 'John' OR name = 'James', type = 'Person') -[e WITH type = 'workAt',
workHours < 40]-> ()
```

Note that the property-accessing expressions in the original value constraints are in-lined into the topology constraint. More specifically, the expressions are in-lined inside the parenthesis or square bracket after the WITH keyword. Moreover, the syntax for property access gets simplified in the in-lined expressions. See the discussion in the following section.

Simplified Property Access in the In-lined Expressions

Syntax for property access is further simplified in the in-lined expressions. In normal value constraint, a property access takes the form of dot expression (i.e. variable_name.property_name). In an in-lined expression, on the other hand, the variable name can be omitted since it is clear from the context. Moreover, if the property name is properly alpha-numeric, even the leading dot can be omitted. The following table summarizes this short-cut rules.

Normal Value Constraint	In-lined Constraint	In-lined Constraint (alternative)
`n.name = 'John'`	`(n WITH .name = 'John')`	`(n WITH name = 'John')`
`n.'middle name' = 'John'`	`(n WITH .'middle name' = 'John')`	

Note that in the above table, we cannot omit the leading dot nor the quotes for property access *'.middle name'* since the name contains a space and is thus not an alpha-numeric.

Nodes/edges without Variable Name but with In-lined Constraints

If a *not-interesting* variable name is omitted for a node or edge term, it is still possible to specify in-lined constraints without having to introduce a variable name. This can be achieved by omitting the variable name and by directly using the WITH keyword followed by the constraints. The following table summarizes this short-cut rule.

In-lined Constraint	In-lined Constraint w/o variable name
`(n WITH name = 'John')`	(WITH name = 'John')

Limitation on the In-lined Expressions

Expressions that contain property accesses from multiple variables (a.k.a. cross-constraints) cannot be in-lined. For example, the following constraints

```
n->m
n.name = m.name
```

cannot be in-lined as

```
(n WITH name = m.name) -> m   // this is a syntax error
```

Identifier short-cut for in-lined expressions

In Property Graph, nodes and edges can have unique identifiers (ID). PGQL expression provides a special syntax for identifier, the built-in function id(). However, there is another short-cut syntax for an in-lined expression, if the node (or edge) is constrained to have a specific ID value. Specifically, the variable name followed by '@' and a certain value means that the node (or edge) should have the ID of the specified value. See the following examples.

Original Syntax	Shortcut Syntax
`(n WITH id() = 123)`	`(n@123)`
`()-[e WITH id()=1234)->[]`	`() -[e@1234]- ()`

Graph Pattern Matching Semantic

There are two popular graph pattern matching semantics: graph homomorphism and graph isomorphism. PGQL's default semantics is based on isomorphism. However, an implementation of PGQL may optionally provide the user with the possibility to set the matching semantic to homomorphism if this is required for a certain use case.

Graph Homomorphism

Under graph homomorphism, multiple nodes (or edges) in the query pattern may match with the same node (or edge) in the data graph as long as all topology and value constraints of the different query nodes (or edges) are satisfied by the data node (or edge).

Consider the following example graph and query:

```
Node 0
Node 1
Edge 0: 0 -> 0
Edge 1: 0 -> 1
```

```
SELECT x, y
WHERE x -> y
```

Under graph homomorphism semantic the output of this query is as follows:

x	y
0	0
0	1

Note that in case of the first result, both query node x and query node y bind to the same data node 0.

Graph Isomorphism

Under graph isomorphism, two query nodes may not match with the same data node.

Consider the following example graph and query:

```
Node 0
Node 1
Edge 0: 0 -> 0
Edge 1: 0 -> 1
```

```
SELECT x, y
WHERE x -> y
```

Under graph isomorphism semantic the output of this query is as follows:

x	y
0	1

Note that x = 0, y = 0 is not a valid solution as x and y may not match with the same data node.

Graph Isomorphism and Multigraphs

Graph isomorphism does not define a semantic for matching query edges with data edges in the case of a multigraph (note: a multigraph is a graph that can have multiple edges between two nodes). This means that it is not defined whether two query edges with the same source and target nodes can match with the same data edge.

SELECT Clause

In a PGQL query, the SELECT clause defines the data entities to be returned in the result. In other words, the select clause defines the columns of the result table.

The following explains the syntactic structure of SELECT clause.

```
<SELECT CLAUSE> := SELECT <select term> (, <select term>)*
                 | SELECT *
<select term> := <expression> (AS <variable>)?
```

A SELECT clause consists of the keyword SELECT followed by a comma-separated sequence of select terms, or a special character star *. A select term consists of:

- An expression.
- An *optional* variable definition that is specified by appending the keyword AS and the name of the variable.

SELECT Expressions

A PGQL query can dictate the data entities to be returned in the SELECT clause, by putting a comma-separated list of expressions after the SELECT keyword. Per every matched subgraph (i.e. row), each SELECT expression (i.e. column) is computed and stored in the result set. For instance, consider the following example:

```
SELECT n, m, n.age
WHERE
   (n WITH type = 'Person') -[e WITH type='friendOf']-> (m WITH type = 'Person')
```

Per each matched subgraph, the query returns two nodes n and m and the value for property *age* of node n. Note that edge e is omitted from the result, even though it is used for describing the pattern.

Assigning Variable Name to Select Expression

It is possible to assign a variable name to any of the selection expression, by appending the keyword AS and a variable name. The variable name is used as the *column name* of the result set. In addition, the variable name can be later used in the ORDER BY clause. See the related section la ter in this document.

```
SELECT n.age*2 - 1 AS pivot, n.name, n
WHERE
   (n WITH type = 'Person') -> (m WITH type = 'Car')
ORDER BY pivot
```

SELECT *

SELECT * is a special SELECT clause. The semantic of SELECT * is to select all the variables or group keys in-scope. If the query has no GROU P BY, the selected variables are all the node and edge variables from the WHERE clause. If the query does have a GROUP BY, the selected elements are all the group keys.

Consider the following query:

```
SELECT *
WHERE
  (n WITH type = 'Person') -> (m) -> (w)
  (n) -> (w) -> (m)
```

Since this query does not have a GROUP BY, all the variables in the WHERE are returned: n, m and w. However, the order of variables selected by SELECT * is not defined by the specification. Therefore the result of SELECT * in the above query can be any combination of (n, m, w).

Now consider the following query, which has a GROUP BY:

```
SELECT *
WHERE
  (n WITH type = 'Person') -> (m) -> (w)
  (n) -> (w) -> (m)
GROUP BY n.name, m
```

Because the query has a GROUP BY, all group keys are returned: n.name and m. The order of the variables selected is the order in which the group keys appear in the GROUP BY.

SELECT * with no variables in the WHERE clause

It is semantically valid to have a SELECT * in combination with a WHERE clause that has not a single variable definition. In such a case, the result set will still contain as many results (i.e. rows) as there are matches of the subgraph defined by the WHERE clause. However, each result (i.e. row) will have zero elements (i.e. columns). The following is an example of such a query.

```
SELECT *
WHERE
  (WITH type = 'Person') -> () -> ()
```

Aggregation

Instead of retrieving all the matched results, a PGQL query can choose to get only some aggregated information about the result. This is done by putting aggregations in SELECT clause, instead of normal expressions. Consider the following example query which returns the average value of property age over all the matched node m.

Aggregation

```
SELECT AVG(m.age) WHERE (m WITH type = 'Person')
```

Syntactically, an aggregation takes the form of Aggregate operator followed by expression inside a parenthesis. The following table is the list of Aggregate operators and their required input type.

Aggregate Operator	Semantic	Required Input Type
COUNT	counts the number of times the given expression has a bound.	not null
MIN	takes the minimum of the values for the given expression.	numeric
MAX	takes the minimum of the values for the given expression.	numeric
SUM	sums over the values for the given expression.	numeric
AVG	takes the average of the values for the given	numeric

COUNT(*) is a special syntax to count the number of matched results, without specifying extra expression. Check the following example:

```
SELECT COUNT(*)
WHERE (m WITH type='Person') -> (k WITH type = 'Car') <- (n WITH type = 'Person')
```

The above simply returns the number of matched subgraphs.

Aggregation and Required Input Type

In PGQL, aggregation is performed only for the matched results where the type of the target expression matches with the required input type. Consider an example Property Graph instance which has the following four node entities.

```
{"id": 3048,  "name":"John",  "age":30}
{"id": 1197,  "name":"Peter", "age":20}
{"id": 20487, "name":"Paul",  "age":"thirty five"}
{"id": 2019,  "name":"James"}
```

Now suppose the following query is applied on this data set.

```
SELECT AVG(n.age), COUNT(*) WHERE (n)
```

Note that all the nodes are matched by the WHERE clause. However, the aggregation result from SELECT clause is 25 and 4. For AVG(n.age) aggregation, only two nodes get aggregated ("John" and "Peter") – the node for "Paul" is not applied because 'age' is not numeric type, and the vertex for "James" does not have 'age' property at all. For COUNT(*) aggregation, on the other hand, all the four matched nodes are applied to the aggregation.

Aggregation and Solution Modifier

Aggregation is applied only after GROUP BY operator is applied, but before the LIMIT and OFFSET operators are applied.

- If there is no GROUP BY operator, the aggregation is performed over the whole match results.
- If there is a GROUP BY operator, the aggregation is applied over each group.

See the detailed syntax and semantics of SOLUTION MODIFIER in the related section in this document.

Assigning Variable Name to Aggregation

Like normal selection expression, it is also possible to assign variable name to aggregations. Again this is done by appending the key word AS and a variable name next to the aggregation. The variable name is used as the *column name* of the result set. In addition, the variable name can be later used in the ORDER BY clause. See the related section later in this document.

```
SELECT AVG(n.age) AS pivot, COUNT(n)
WHERE
   (n WITH type = 'Person') -> (m WITH type = 'Car')
GROUP BY n.hometown
ORDER BY pivot
```

Solution Modifier Clause

The SOLUTION MODIFIER clause defines additional operations for building up the result of the query. A SOLUTION MODIFIER clause consists of four (sub-)clauses– <GROUP BY clause>, <ORDER BY clause>, <LIMIT clause> and <OFFSET clause>. Note that all these clauses are optional; therefore the entire SOLUTION MODIFIER clause is optional.

```
<SOLUTION MODIFIER Clause> :=
    (<GROUP BY clause>)?
    (<ORDER BY clause>)?
    (<LIMIT-OFFSET clauses>)?
```

ORDER BY

When there are multiple matched subgraph instances to a given query, in general, the ordering between those instances are not defined; the query execution engine can present the result in any order. Still, the user can specify the ordering between the answers in the result using

ORDER BY clause.

The following explains the syntactic structure of ORDER BY clause.

```
<ORDER BY clause> := ORDER BY <order term> (, <order term>)*

<order term> :=  <expression> (ASC|DESC)
              |  (ASC|DESC) '(' <expression> ')'
```

The ORDER BY clause starts with the keywords ORDER BY and is followed by comma separated list of order terms. An order term consists of the following parts:

- An expression.
- An *optional* ASC or DESC decoration to specify that ordering should be ascending or descending.
  - If no keyword is given, the default is ascending order.

The following is an example in which the results are ordered by property access *n.age* in ascending order.

```
SELECT n.name
WHERE (n WITH type = 'Person')
ORDER BY n.age ASC
```

Multiple Terms in ORDER BY

It is possible that ORDER BY clause consists of multiple terms. In such a case, these terms are evaluated from left to right. That is, (n+1)th ordering term is used only for the tie-break rule for n-th ordering term. Note that each term can have different ascending or descending decorator.

```
SELECT f.name
WHERE (f WITH type = 'Person')
ORDER BY ASC(f.age), f.salary DESC
```

Data Types for ORDER BY

A partial ordering is defined for the different data types as follows:

- Numeric data values are ordered from small to large.
- Strings are ordered lexicographically.
- Nodes and edges are ordered by their identifier (small to larger if numeric, lexicographically if String)

In the case a property access holds multiple types of data values, the following ordering is applied between values of different types:

- Numeric < String < Boolean 'true' < Boolean 'false' < 'null'

Consider the following data values:

```
['Mary', 25, null, true, false, 'John', 3.5, 27.5]
```

Applying the above rules to the values, will result in the following ordering:

```
[3.5, 25, 27.5, 'John', 'Mary', true, false, null]
```

LIMIT and OFFSET

The LIMIT puts an upper bound on the number of solutions returned, whereas the OFFSET specifies the start of the first solution that should be returned.

The following explains the syntactic structure for the LIMIT and OFFSET clauses:

```
<LIMIT-OFFSET clauses> := (LIMIT <integer>) (OFFSET <integer>)?
                        | (OFFSET <integer>) (LIMIT <integer>)?
```

The LIMIT clause starts with the keyword LIMIT and is followed by an integer that defines the limit. Similarly, the OFFSET clause starts with the keyword OFFSET and is followed by an integer that defines the offset. Furthermore:

- The LIMIT and OFFSET clauses can be defined in either order.
- The limit and offset may not be negatives.

The following semantics hold for the LIMIT and OFFSET clauses:

- The OFFSET clause is always applied first, even if the LIMIT clause is placed before the OFFSET clause inside the query.
- An OFFSET of zero has no effect and gives the same result as if the OFFSET clause was omitted.
- If the number of actual solutions after OFFSET is applied is greater than the limit, then at most the limit number of solutions will be returned.

Example:

In the following query, the first 5 intermediate solutions are pruned from the result (i.e. OFFSET 5). The next 10 intermediate solutions are returned and become final solutions of the query (i.e. LIMIT 10).

```
SELECT n WHERE (n) LIMIT 10 OFFSET 5
```

Grouping and Aggregation

GROUP BY allows for grouping of solutions and is typically used in combination with aggregation to aggregate over groups of solutions instead of over the total set of solutions.

The following explains the syntactic structure of the GROUP BY clause:

```
<GROUP BY clause> := GROUP BY <group term> (, <group term>)*
<group term>      := <expression> (AS <variable>)?
```

The GROUP BY clause starts with the keywords GROUP BY and is followed by a comma-separated list of group terms. Each group term consists of:

- An expression.
- An *optional* variable definition that is specified by appending the keyword AS and the name of the variable.

Consider the following query:

```
SELECT n.firstName, COUNT(*), AVG(n.age) WHERE (n WITH type = 'Person') GROUP BY
n.firstName
```

Matches are grouped by their values for *n.firstName*. For each group, the query selects *n.firstName* (i.e. the group key), the number of solutions in the group (i.e. *COUNT(*)*), and the average value of the property age for node n (i.e. *AVG(n.age)*).

Assigning Variable Name to Group Expression

It is possible to assign a variable name to any of the group expression, by appending the keyword AS and a variable name. The variable name can be used in the SELECT to select a group key, or in the ORDER to order by a group key. See the related section later in this document.

```
SELECT nAge, COUNT(*)
WHERE
   (n WITH type = 'Person')
GROUP BY n.age AS nAge
ORDER BY nAge
```

Multiple Terms in GROUP BY

It is possible that GROUP BY clause consists of multiple terms. In such a case, matches are grouped together only if they hold the same result for each of the group expressions.

Consider the following query:

```
SELECT n.firstName, n.lastName, COUNT(*) WHERE (n WITH type = 'Person') GROUP BY
n.firstName, n.lastName
```

Matches will be grouped together only if they hold the same values for *n.firstName* and the same values for *n.lastName*.

GROUP BY and NULL values

The group for which all the group by expressions evaluate to null is ignored and does not take part in further query processing. However, a group for which some expressions evaluate to null but at least one expression evaluates to a non-null value, is *not* ignored and takes part in further query processing.

Variable Definition, Access, and Visibility

Variables may be defined in the SELECT, WHERE and GROUP BY clauses and may be accessed in the SELECT, WHERE, GROUP BY and ORDER BY clauses. However, there are certain visibility rules that describe whether a variable is visible in a certain context and whether it may thus be accessed or not. The table below gives an overview of the rules.

	Variable definition and access	**Variable visibility rules**
SELECT	Variable definition in the SELECT takes the form of a variable assignment to a select expression (i.e. exp AS var). Variable access may happen in the expression part of a select expression.	The visibility rules depend on whether a GROUP BY is defined or not: • Query has *no* GROUP BY: ◦ Variables defined in WHERE are visible. • Query has GROUP BY: ◦ Variables defined in GROUP BY are visible, both inside and outside aggregations (i.e. SELECT var, AVG(var)). ◦ Variables defined in WHERE are visible, but only inside aggregations (i.e. SELECT AVG(var)).
WHERE	Variable definitions in the WHERE are the named nodes and edge in the topology constraints. (e.g *n1* and *e1* in *n1 -[e1]-> ()*). Variable access may happen in a value constraint.	The variable that are visible in the WHERE are those defined in the WHERE (i.e. all the node variables and edge variables).
GROUP BY	Variable definition in the GROUP BY takes the form a of a variable assignment to a group expression (i.e. exp AS var). Variable access may happen in the expression part of a group expression.	The variable that are visible in the GROUP BY are those defined in the WHERE (i.e. all the node variables and edge variables).

ORDER BY	The ORDER BY does not introduce new variables (i.e. *exp* is allowed, but *exp AS var* is not). Variable access may happen in an order by expression.	The access rules depend on whether a GROUP BY is defined or not: • Query has *no* GROUP BY: • Variables defined in SELECT are visible. • Variables defined in WHERE are visible. • Query has GROUP BY: • Variables defined in GROUP BY are visible, both inside and outside aggregations (i.e. ORDER BY var, AVG(var)). • Variables defined in SELECT and WHERE are visible, but only inside aggregations (i.e. ORDER BY AVG(var).

Repetition of Group Expression in Select or Order Expression

Group expressions that are variable accesses, property accesses, or built-in function calls may be repeated in select or order expressions. This is a short-cut that allows you to neglect introducing new variables for simple expressions.

Consider the following query:

```
SELECT n.age, COUNT(*)
WHERE
  n
GROUP BY n.age
ORDER BY n.age
```

Here, the group expression n.age is repeated as select and order expressions.

This repetition of group expressions introduces an exception to the variable visibility rules described above, since variable n is not inside an aggregation in the select/order expression. However, semantically, the query is treated as if there were a variable for the group expression:

```
SELECT nAge, COUNT(*)
WHERE
  n
GROUP BY n.age AS nAge
ORDER BY nAge
```

Expressions

Expressions are used in value constraints, in-lined constraints, and select/group/order terms. This section of the document defines the operators and built-in functions that can be used as part of an expression.

Operators

The following table is an overview of the operators in PGQL.

Operator type	**Operator**	**Example**
Arithmetic	+, -, *, /	SELECT * WHERE n -> (m WITH start_line_num < end-line-num - 10)
Relations	=, !=, <, >, <=, >=	SELECT * WHERE n --> m, n.start_line_num < m.start_line_num
Logical	AND, OR, NOT, !	SELECT * WHERE n --> m, n.start_line_num > 500 AND m.start_line_num > 500

Operator and Operand Types

The following table specifies operand types and operator return types.

Operator	Type(A)	Type(B)	Result type
A + B A - B A * B A / B	numeric	numeric	numeric
A = B A != B A < B A > B A <= B A >= B	numeric	numeric	boolean
A = B	String	String	boolean
A = B	boolean	boolean	boolean
A AND B A OR B	boolean	boolean	boolean
NOT A !A	boolean		boolean

If the value for an operand is of a type that is not defined for the operator, the operation yields *null*. There is one exception to this rule, which is that the OR operator yields true if either of the operands yield true (see the section on null values and operators).

Data Type Conversion

Numeric values are automatically converted (coerced) when compared against each other. An example is as follows:

```
3 = 3.0 // this expression yields TRUE
```

Comparing between Numeric, String, or Boolean values yields null (see the next section for more details on the handling of null values).

Null Values

'null' is used to represent a missing or undefined value. There are three ways in which a null value can come into existence:

- A property access (i.e. var_name.prop_name) returns null if the property is missing for a node or edge in the data graph.
- An expression returns null if any operand or function argument is null (with an exception for the OR operator, see below).
- In a query, a 'null' value may be used in the place of a literal value (e.g. n.name = null).

Null Values and Operators

An operator returns null if one of its operands yields null, with an exception for the OR operator: if the left-hand side or right-hand side of the OR operations returns true, the operation itself yields true. Otherwise, the operation yields null. The table below summarizes these rules.

Operator	Result {A = NULL}	Result {B = NULL}	Result {A = NULL, B = NULL}
A + - * / B A = != < > <= >= B	null	null	null
A AND B	null	null	null
A OR B	true if B yields true, null otherwise	true if A yields true, null otherwise	null
NOT A !A	null		

Note that from the table it follows that *null = null* yields *null* and not *true*. Not knowing two values does not imply that they are the same value.

Null Values as Function Argument

If any of the arguments of a function is null, the function itself yields null. For example, *x.has(null)* yields null.

Built-in Functions

Built-in functions can be used in a value constraint or an in-lined constraint, or in a select/group/order expression. The following table lists the built-in functions of PGQL.

Object type	Signature	Return value	Description	Notes
node/edge	id()	numeric/string	returns the node/edge identifier	
node/edge	has(prop1, prop2, ...)	boolean	returns true if the node or edge has the given (comma-separated) properties.	The arguments need to follow the syntax for properties (i.e. quotes can optionally be omitted if the property name is an alphanumeric)
node	inDegree()	decimal	returns the number of incoming neighbors.	
node	outDegree()	decimal	returns the number of outgoing neighbors.	

The syntactic structure of a built-in function call is as follows:

```
<function call> := <name> '(' <function argument>* ')'
```

A build-in function call is a function name followed by zero or more function arguments. The function arguments are in between rounded brackets. Furthermore, function names are *not* case-sensitive.

In contrast to SQL, the node or edge to which the function applies (i.e. the *object*), is not passed as one of the function arguments. Instead, the same dot expression syntax that is used for a property access, is also used for a function call: *variable_name.function_name(function arguments)*. Consider the following example query:

```
SELECT y.id()
WHERE
  x -> y,
  x.inDegree() > 10
```

Here, *x.inDegree()* returns the number of incoming neighbors of x, whereas *y.id()* returns the identifier of the node y. Variables x and y are the objects of the two function calls.

Simplified Function Calls in the In-lined Expressions

The same syntactic structure rules that apply to a simplified property access, also apply to a function call in an in-lined expression. That is, the object of the function call can be omitted since it is clear from the context. Moreover, the leading dot can be omitted too. The following table summarizes these short-cut rules.

Normal Function Call	**In-lined Function Call**	In-lined Function Call (alternative)
`n.outDegree() > 10`	`(n WITH .outDegree() > 10)`	`(n WITH outDegree() > 10)`

Other Syntactic Rules

Syntax for Variables

The syntactic structure of a variable name is an alphabetic character followed by zero or more alphanumeric or underscore (i.e. _) characters:

```
<variable> := [a-zA-Z][a-zA-Z0-9\_]*
```

Syntax for Properties

Property names may be quoted or unquoted. Quoted and unquotes property names may be used interchangeably. If *unquoted*, the syntactic structure of a property name is the same as for a variable name. That is, an alphabetic character followed by zero or more alphanumeric or underscore (i.e. _) characters. If *quoted*, the syntactic structure is that of a String (for the syntactic structure, see String literal).

```
<property name> := [a-zA-Z][a-zA-Z0-9\_]*
                 | <string>
```

Literals

The literal types are String, Integer, Decimal, and Boolean. The following shows the syntactic structure of the different types of literals.

Literal type	**Syntax**	**Example**

String	`<string> := "'" (~[\'\n\\] | <escaped character>)* "'"`
	`          | '"' (~[\"\n\\] | <escaped character>)* '"'`
		'Person'
		"Person"

Integer	`<integer> := [0-9]+`	25
Decimal	`<decimal> := [0-9]* '.' [0-9]+`	17.3 .4
Boolean	`<boolean> := 'true' | 'false'` Boolean literals are case-insensitive.	true false

Single-quoted and Double-quoted Strings

A String literal may either be single or double quoted. Single and double quoted Strings can be used interchangeably. For example, the following expression evaluates to *true*.

```
"Person" = 'Person' // this expression evaluates to TRUE
```

Escaped Characters in Strings

Escaping in String literals is necessary to support having white space, quotation marks and the backslash character as a part of the literal value. The following explains the syntax of an escaped character.

```
<escaped character> := '\' [tnr\"']
```

Note that an escaped character is either a tab (\t), a line feed (\n), a carriage return (\r), a single (\') or double quote (\"), or a backslash (\\). Corresponding Unicode code points are shown in the table below.

Escape	**Unicode code point**
\t	U+0009 (tab)
\n	U+000A (line feed)
\r	U+000D (carriage return)
\"	U+0022 (quotation mark, double quote mark)
\'	U+0027 (apostrophe-quote, single quote mark)
\\	U+005C (backslash)

Optional Escaping of Quotes in Strings

In single quoted String literals, it is optional to escape double quotes, while in double quoted String literals, it is optional to escape single quotes. The following table provides examples of String literals with escaped quotes, and corresponding String literals in which quotes are not escaped.

With escape	**Without escape**
'single quoted string literal with \"double\" quotes inside'	'single quoted string literal with "double" quotes inside'
"double quoted string literal with \'single\' quotes inside"	"double quoted string literal with 'single' quotes inside"

Note that the value of the literal is the same no matter if quotes are escaped or not. This means that, for example, the following expression evaluates to TRUE.

```
'\"double" quotes and \'single\' quotes' = "\"double\" quotes and \'single' quotes" //
this expression evaluates to TRUE
```

Keywords

The following is the list of keywords in PGQL.

```
SELECT, WHERE, AS, WITH, ORDER, GROUP, BY, ASC, DESC, LIMIT, OFFSET, AND, OR, true,
false, null
```

There are certain restrictions when using keywords as variable or property name:

- Keywords cannot be used as a variable name.
- Keywords can only be used as a property name, if quotation for property name access is used:

```
SELECT n
WHERE (n)->(m WITH type='Employee')
  n.'GROUP' = 'managers'
```

Finally, keywords are *not* case-sensitive. For example, SELECT, Select and sELeCt, are all valid.

Comments

There are two kinds of comments: single-line comments and multi-line comments. Single-line comments start with double backslashes (\\), while multi-line comments are delimited by /* and */. The following shows the syntactic structure of the two forms.

```
<single-line comment> := '//' ~[\n]*

multi-line comment> := '/*' ~[\*]* '*/'
```

An example query with both single-line and multi-line comments is as follows:

```
/* This is a
   multi-line
   comment */
SELECT n.name, n.age
WHERE
  (n WITH type = 'Person') // this is a single-line comment
```

White Space

White space consists of spaces, new lines and tabs. White space is significant in String literals, as the white space is part of the literal value and taken into account when comparing against data values. Outside of String literals, white space is ignored. However, for readability consideration and ease of parser implementation, the following rules should be followed when writing a query:

- A keyword should not be followed directly by a variable or property name.
- A variable or property name should not be followed directly by a keyword.

If these rules are not followed, a PGQL parser may or may not treat it as an error.

Consider the following query:

```
SELECT n.name, m.name
WHERE
  (n WITH type = 'Person', name = 'Ron Weasley') -> m
```

This query can be reformatted with minimal white space, while guaranteeing compatibility with different parser implementations, as follows:

```
SELECT n.name,m.name WHERE(n WITH type='Person',name='Ron Weasley')->m
```

Note that the white space after the SELECT keyword, in front of the WHERE keyword, before and after the WITH keyword and in the String literal 'Ron Weasley' cannot be omitted.

- Variables defined in WHERE are only visible inside aggregations unless a group expression exists for the variable access, in which case the variable is also visible outside aggregations.

There is also an exception that allows for repeating simple group expression in a select or order by expression without having to introduce a new variable.

- An expressions.