Title: Path Pattern Union

Continuing on [JCJ-029] and responding to [sql-pg-2019-0033]

Source: Individual Experts Contribution
Authors: Neo4j Query Languages Standards and Research Team
Status: Discussion paper

Abstract

We propose an operator in path patterns expressed using a vertical bar character (" | ") called *path pattern union*. This operator expresses a choice between alternative patterns (the operands of the operator). We propose that the *path pattern union* operator replaces the previously proposed operators *Multiset Alternation* and *Set Disjunction*. This proposal describes in the semantics of the *path pattern union* operator in order to facilitate collective understanding of the topic and provide a base for further discussion. It is not yet in the form needed by standard. We show how the semantics of *Multiset Alternation* may trivially be achieved, as well as how these semantics extend to *Path macros*.

When syntax productions are described in this paper, this is intended as a reminder to the reader, to make it easier to read without having to reference the definition. For that reason, such production rules have been shortened and simplified in this paper, which means that they may not always be complete or perfectly correct, but should still be able to serve their illustrative purpose.

---

1 Hannes Voigt, Petra Selmer, Stefan Plantikow, Tobias Lindaaker, Alastair Green, Peter Furniss.
Table of contents

1. References 3

2. Proposal 4

3. Background on identity and sets 5

4. Overview of the pattern language 6
   4.1. Vertex patterns 6
   4.2. Edge patterns 7
   4.3. Concatenation path joining 8
   4.4. Juxtaposition path joining 8
   4.5. Path patterns 9
   4.6. Cartesian products 10
   4.7. Natural joins 10
   4.8. Path pattern union 11
   4.9. Quantification 11
      4.9.1. Semantic equivalence of \{n\} 12
      4.9.2. Semantic equivalence of \{n,m\} 13
      4.9.4. Semantics of the empty pattern, denoted by \varepsilon 14
   4.10. CHEAPEST and SHORTEST 15

5. Variable assignment and pattern canonicalization 16
   5.1. Canonicalization of patterns 18
   5.2. Path pattern variable assignment 19

6. Path pattern union 20
   6.1 Fixed patterns 21
      Example 1: Positionally compatible elements with identical explicit variables 21
      Example 2: Positionally compatible elements with different explicit variables 22
      Example 3: Positionally compatible elements, where one is implicit, and the other explicit 22
      Example 4: Positionally compatible elements with identical implicit variables 24
      Example 5: Achieving Multiset Alternation 24
      Example 6: Considering path variables 26
      Example 7: Operands of differing lengths 29
   6.2 Quantified patterns 31
      Example 8: Canonicalization of quantified patterns 31

7. Path macros 35
1. References


[YTZ-036] Fred Zemke, “Regular expression syntax for graph patterns”, May 2018


[JCJ-015] Fred Zemke, “[+] for multiset alternation”, May 2019


[JCJ-029] Neo4j Query Languages Standards and Research Team, “Unifying ‘Set Disjunction’ and ‘Multiset Alternation’”, June 2019
2. Proposal

We propose an operator in path patterns expressed using a vertical bar character (" | ") called **path pattern union**\(^2\). This operator expresses a choice between alternative patterns (the operands of the operator). We propose that the *path pattern union* operator replaces the previously proposed operators *Multiset Alternation* and *Set Disjunction*.

The intention of this proposal is to describe in detail the semantics of the *path pattern union* operator, thereby hopefully facilitating the collective understanding of this topic by the group.

The semantics of *path pattern union* are best explained by first transforming the syntactic form of the operands into a canonical form consisting of *fixed patterns*, where each *element pattern* and *path pattern* are assigned at least one variable. If no variables are explicitly provided in the syntactic form, the algorithm described in this paper will assign *implicit variables* so that element or path patterns in the same position in different operands of *path pattern union* are assigned the same implicit variable.

Once variables have been assigned to all element and path patterns of the fixed pattern operands of *path pattern union*, the semantics (of *path pattern union*) are that of the regular relational set union between the tables produced by matching the fixed patterns of the operands, where each of these tables are extended to have columns matching the complete set of variables from any operand ("outer union").

The semantics of "outer union" can be implemented without having to materialise these tables and deduplicate the rows. Given complete sets of canonical (fixed-length) operands for all operands, it is straightforward to recognize when two operands produce the same bindings for the same positions and combine the predicates of these into a single operand.

If the user desires the semantics of *Multiset Alternation*, this can be achieved easily by using different explicit variables in different operands. This can frequently be done without being obtrusive by for example introducing an explicit path variable for a subpath.

The algorithm for assigning implicit variables is further useful for ensuring hygienic expansion of *Path macros*. We show how this solves the commutativity problem of alternatives between *Path macros* and label predicate expressions described in [YTZ-036] (Section 6.3.1, Note 3).

\(^2\) We take on board the criticism raised in [sql-pg-2019-0033] that ‘union’ as a term is ambiguous and thereby leads to confusion. In the absence of a better term, we’ve used instead ‘*path pattern union*’ in this paper simply as a convenience, and welcome other suggestions.
3. Background on identity and sets

As a prelude to the following sections, we wanted to give some background around graphs and sets (in response to Section 3.1 of [sql-pg-2019-0033]).

Each element in a graph - i.e. a vertex or an edge - has intrinsic identity. It is the case that two vertices \( v_1 \) and \( v_2 \) may be different vertices even though \( v_1 \) has the same labels and properties as \( v_2 \). The notion of these vertices being distinct vertices in their own right is of particular importance because the vertices in question may be connected by different sets of edges. This applies to edges in an analogous manner (for example, in cases where the user wishes to model the fact that the same type of interaction occurred multiple times for the same endpoints).

This notion of “topology first” is further strengthened by the case that we may have graphs containing elements having neither properties nor labels (or labels, but no properties). In these graphs, the connectivity patterns between the vertices and edges essentially acts as the data. It is clear that each element will need to have intrinsic identity.

This means that the vertices and edges, when considering their intrinsic identity, do indeed form sets - and is in line with the classic mathematical definition of a graph. This holds even if a tabular projection of the “values” (labels and properties) of vertices and edges would form a multiset.

Building upon the foundational sets that make up the graph, matching of “fixed patterns” (without quantification or path pattern union) results in sets in the identities of the underlying elements (when considering all parts of the pattern).

In this paper we argue that path pattern union (and quantification, which builds upon it) should also result in sets in the same way, so that the pattern language becomes semantically composable.

---

3 Except in the case where neither of them are connected by any edges.

4 An example of a domain where this mode is used is in the modelling of water pipes and junction points in the civic utilities domain.
4. Overview of the pattern language

In this section we present an overview of the building blocks available in the pattern language. We will mention path pattern union, the primary topic of this paper, but give it full treatment in Section 5. This expands on Section 4 of [JCJ-029] “Path Patterns and Graph Patterns”.

Although this section restates things that have appeared in previous papers, we do so in order to provide a solid and coherent foundation for the purposes of motivating our proposal.

4.1. Vertex patterns

As detailed in [SQL/PGQ 8IWD2], a Vertex pattern expresses:

- a variable\(^5\) used to identify the pattern and to reference vertices matched by the pattern,
- a label expression describing the labels a matching vertex must have, and
- a set of general predicates that a matching vertex must satisfy (typically these are predicates on the properties of the vertex).

While the label expressions and predicates are syntactically distinguished from one another, semantically they form a single set of predicates that a matching vertex must satisfy. From a semantic perspective we thus think of the Vertex pattern as being constituted by a vertex variable, with predicates expressed over that variable.

Vertex patterns and Edge patterns form the set of Element patterns that are used for matching graph elements.

A Vertex pattern is a Path pattern of length zero (0), with matches consisting only of a single vertex.

A short and simplified syntax summary of Vertex patterns is:

\[
\text{<vertex pattern> ::= '('
[<vertex pattern variable>]
[('IS' | ':') <label expression>]
['WHERE' <search condition>]
')']
\]

and an example would be:

\[(bob IS Person WHERE bob.name = 'Bob')\]

---

\(^5\) We note, tangentially, that the syntax around the declaration of variables in patterns has a redundant indirection in its declaration, with

\[
\text{<graph pattern variable declaration> ::= '<graph pattern variable>}
\]
4.2. Edge patterns

Edge patterns, together with Vertex patterns, constitute the Element patterns for matching the elements of the graph and from which other patterns are formed.

An Edge pattern, as detailed in [SQL/PGQ 8IWD2], expresses:

- a variable used to identify the pattern and to reference edges matched by the pattern,
- a label expression describing the labels that a matching edge must have,
- a set of general predicates that a matching edge must satisfy, and
- the Vertex patterns that vertices connected by the edge pattern must satisfy.

Edge patterns, by connecting two Vertex patterns, specify joins between matching vertices and the matching edge in such a way that:

- the vertices matching the source Vertex pattern of the Edge pattern are joined with the source vertex of edges matching the Edge pattern on the identity of the vertex, and
- the vertices matching the target Vertex pattern of the Edge pattern are joined with the target vertex of edges matching the Edge pattern on the identity of the vertex.

A short and simplified syntax summary of Edge patterns is:

```
<edge pattern> ::= 
    ( <source vertex pattern> 
      ( '->' | '-[<edge filler> ]->' | '-' <mandatory filler> '->' ) 
      <target vertex pattern> ) 
  | ( <target vertex pattern> 
      ( '<-' | '<-[<edge filler> ]-' | '<-' <mandatory filler> '-' ) 
      <source vertex pattern> ) 
  | ( <vertex pattern> 
      ( '-' | '-[<edge filler> ]-' | '-' <mandatory filler> '-' ) 
      <vertex pattern> )

<edge filler> ::= 
    [<edge pattern variable>] 
    [[('IS' | ':') <label expression>]] 
    ['WHERE' <search condition>]

<mandatory filler> ::= 
    <edge pattern variable> [[('IS' | ':') <label expression>]] 
    | ('IS' | ':') <label expression>

<source vertex pattern> ::= <vertex pattern> 
<target vertex pattern> ::= <vertex pattern>
```
An example would be:

\[(\text{bob})-\text{[ e : is_a WHERE e.since = 'forever' ]}-(\text{festis})^{6}\]

An **Edge pattern** is a **Path pattern** of length one (1), with matches consisting of a single edge in direction from left to right in the syntactic expression of the pattern.

Within a **Path pattern** it is allowed to abbreviate an edge pattern that does not specify any explicit constraints about its source and target vertices by omitting the source vertex pattern and the target edge pattern.

### 4.3. Concatenation path joining

Two **Path patterns** may be combined to form a **Path pattern** by the sharing one **Vertex pattern**, such as in this example:

\[(\text{a})-\text{[e]}-(\text{b})-\text{[f]}-(\text{c})\]

which combines the **Path patterns** formed by the **Edge patterns** for ‘e’ and ‘f’ by sharing the **Vertex pattern** for ‘b’.

### 4.4. Juxtaposition path joining

Two **Path patterns** can be combined to form a **Path pattern** by stating them juxtaposed next to one another. The combination of these patterns are formed by joining the adjacent **Vertex patterns** of the juxtaposed **Path patterns** to one another, such as in this example:

\[(\text{a})-\text{[e]}-(\text{b})\ (\text{c})-\text{[f]}-(\text{d})\]

where the **Path patterns** formed by the **Edge patterns** for ‘e’ and ‘f’ are combined through the **Vertex pattern** for ‘b’ being joined with the **Vertex pattern** for ‘c’ (i.e. b and c must be matched to the same vertex in the graph).

As shown in **Section 4.9 (Quantification)**, juxtaposed patterns naturally arise during the course of quantification expansion, and it is useful to describe the semantics (of juxtaposed patterns) before presenting the semantics of quantification. Juxtaposed patterns can also arise with Path patterns as in **Section 4.5**. We are not proposing that it must be possible to express juxtaposed patterns explicitly, but their raw syntactic expression is useful in discussing the semantics.

---

6 This example is a reference to an [ad for a Swedish soft drink](#).
4.5. Path patterns

Path patterns are formed by:

- **Vertex patterns** - forming a pattern for a path of length 0.
- **Edge patterns** - forming a pattern for a path of length 1.
- **Path Patterns** joined by Concatenation.
- **Path Patterns** joined by Juxtaposition.
- **Quantified Path Patterns** - a Path Pattern repeated a number of times in a given range.
- A **Union** of Path Patterns - expressing that a matching path must match one of several alternatives.
- Expressions limiting which qualifying paths to keep. This ranges from a simple WHERE to specify a predicate for the path (described in this section), to the ability to specify that only the shortest paths or the cheapest paths based on a cost function should be accepted (described in Section 4.10).

A Path pattern may furthermore express the binding of a path variable, as well as predicates over all the variables bound by a Path pattern.

Note that due to the compositional nature of Path patterns, one path variable could capture a subpath of another Path pattern.

A simple syntactic expression showing the composition of Path pattern could be:

```
<path pattern> ::= 
  [ <path pattern variable> '=' ]
<path pattern expression>
  [ 'WHERE' <search condition> ]

<path pattern expression> ::= 
  <path term> [{<path pattern union operator> <path term>}...]

<path term> ::= <path factor>...

<path factor> ::= 
  <path primary> [<path pattern quantifier>]

<path primary> ::= 
  <vertex pattern> 
  | <edge pattern>
```

7 See Section 4.8 on Path pattern union.

8 See Section 4.9 on Quantification.
This example shows the use of `WHERE` with a `Path pattern` to express conditions spanning multiple elements:

```
// a path from x to y through progressively older friends
(x) → ([:KNOWS]→(b) WHERE a.age < b.age )* (y)
```

The ability to have `WHERE` associated with a `Path pattern` has real utility, since the `Path pattern` is a context in which elements can be kept as singleton variables, whereas in the `WHERE` of `MATCH`, variables under quantification will be turned into group variables. This is in contrast to `WHERE` in Vertex patterns and Edge patterns, which are purely a syntactic convenience, as these could equally well be expressed in `WHERE` associated with a `Path pattern`.

4.6. Cartesian products

A graph pattern can be composed from multiple `Path patterns` separated by a comma (`,`). The result is the Cartesian product between the paths matching each of the `Path patterns`. Typically these Cartesian products are constrained through `Natural joins` to form conjunctive patterns.

For example, `(a)→(b), (b)→(g)` forms a conjunctive pattern, and `(a)→(b), (c)→(d)` would form a full Cartesian product.

4.7. Natural joins

If the same variable is used in multiple parts of a pattern, this stipulates a `Natural join` between those parts on that variable, where the content of that variable has to be the same between the two paths. An example of this is the following pattern which matches multiple edges from the same vertex:

```
(v)→[IS left_of]→(r), (v)→[IS right_of]→(l),
(v)→[IS above]→(b), (v)→[IS below]→(a)
```

Intuitively, using the same variable in more than one place means that the matching data in all occurrences of that variable `has to be the same` for a valid match.

Since predicates for a pattern may be expressed for any occurrence of the same variable, the predicates for that variable are combined conjunctively, except if the two occurrences are separated by a `path pattern union` (`|`), in which case the predicates are effectively combined disjunctively (see Example 8).

Natural joins can be expressed within a single `Path pattern` through the repetition of a variable within the `Path pattern`. This means that a matching path has to revisit the same element, i.e. match that element in more than one place of the path. The simplest example of this is a pattern...
that matches a loop (an edge that starts and ends at the same vertex) which is expressed through an edge pattern that repeats the same vertex pattern variable as both the source and target:

\[(v) - [\text{loop}] - (v)\]

More complex examples are of course possible, such as describing a cycle of alternating RED and BLUE edges:

\[(v) - [:\text{RED}] - [:\text{BLUE}] - (v)\]

It is also possible to repeat other kinds of variables, such as Edge pattern variables:

\[(a) - [e: \text{Knows}] - (b), \ (x) - [e] - (y)\]

This has the side effect of joining the ‘x’ vertices with the ‘b’ vertices and joining the ‘y’ vertices with the ‘a’ vertices. Since the ‘e’ edge has to be the same in both places it has to have the same source vertices and the same target vertices in both places, thus the ‘a’ vertices have to be the same as the ‘y’ vertices and the ‘b’ vertices have to be the same as the ‘x’ vertices.

It is well worth clarifying how natural joins relate to singleton and group variables [BNE-034]⁹. The semantics described in this section pertain only to singleton variables. This means that:

- A singleton variable may not be joined with a group variable. This applies to variables that are singleton vs group within the same scope¹⁰.
- A group variable may not be joined with another group variable. For example, this would be forbidden (x is the variable under consideration):

\[((a) - (x) - (b))* - ((m) - (x) - (n))*\]

4.8. Path pattern union

A path pattern union specifies two or more alternative Path patterns separated by a vertical bar (“|”) operator symbol. We call these alternative path patterns the operands of the path pattern union¹¹ operator (denoted “|”).

The semantics of path pattern union is covered in greater detail in Section 6.

---

⁹ We remind the reader that a variable \(v\) in an inner scope is a singleton variable, and that in an outer scope (outside a quantified pattern), \(v\) is a group variable.

¹⁰ We note that a global singleton is a variable that is not quantified anywhere within the query.

¹¹ We are using the name path pattern union since it communicates the semantics of expressing a union between path patterns. We are open to suggestions for other names.
4.9. Quantification

Path Patterns may be quantified. The canonical syntactic form of a quantifier is

\{n,m\}

which means repeat the quantified path pattern between \(n\) and \(m\) times. If \(n\) is omitted it defaults to \textit{zero}; if \(m\) is omitted it defaults to \textit{infinity}.

In total, the following syntactic forms of quantifications are allowed:

- \* - repeat the quantified path pattern \textit{zero or more} times
- + - repeat the quantified path pattern \textit{one or more} times
- ? - repeat the quantified path pattern \textit{zero or one} times (“optional”)
- \{n,m\} - repeat the quantified path pattern between \(n\) and \(m\) times
  - if \(n\) is omitted it defaults to \textit{zero}
  - if \(m\) is omitted it defaults to \textit{infinity}
- \{n\} - repeat the quantified path pattern exactly \(n\) times

These quantifications adhere to the semantic equivalences described in the following subsections.

4.9.1. Semantic equivalence of \{n\}

\{n\} is semantically equivalent to the pattern repeated \(n\) times.

Given a pattern \(\alpha\), then \(\alpha\{n\}\) is equivalent to:

- the empty pattern (\(\epsilon\)) when \(n = 0\).
- \(\alpha \alpha’\{n-1\}\) when \(n > 0\).
  - \(\alpha’\) denotes the same pattern as \(\alpha\) in which fresh variables have been assigned

Unrolling the recursion fully for a few cases, we arrive at:

- \(\alpha\{0\}\) expands to \(\epsilon\)
- \(\alpha\{1\}\) expands to \(\alpha\)
- \(\alpha\{2\}\) expands to \(\alpha \alpha’\)
- \(\alpha\{3\}\) expands to \(\alpha \alpha’ \alpha’’\)
- ...and so on...

\[12\] Since a pattern expressing things that should be matched exactly zero times is nonsensical, 0-quantification patterns should not be syntactically allowed. See Section 4.9.4.
\(\alpha'\) (and \(\alpha''\), et c.) indicates separate naming contexts, or namespaces, which qualify the names of variables for each repetition of the pattern; i.e. the names of variables in one repetition do not interfere with the names of variables in another repetition. The results bound to the same variables in different naming contexts are however collected together into the same grouped variable on the outside of the pattern.

The same naming context will appear in multiple operands of the implied path pattern union. However, this does not cause any interference, since the patterns in each operand are of different fixed shape and cannot match at the same time.

4.9.2. Semantic equivalence of \(\{n,m\}\)

\(\{n,m\}\) - is considered to be the path pattern union of each fixed repetition of the pattern.

Given a pattern \(\alpha\), then \(\alpha\{n,m\}\) (where \(m\) may be infinity\(^{13}\)) is the path pattern union of all patterns in the set:

\[
\{ \alpha\{k\} \mid n \leq k \leq m \}
\]

In the case where \(m\) is infinite (i.e. unspecified), this leads to an infinite set.

Spelling out the path pattern union of this set (which we cannot do if \(m\) is infinite), we get:

\[
\alpha\{n\} \mid \alpha\{n+1\} \mid \ldots \mid \alpha\{m-1\} \mid \alpha\{m\}
\]

For a pattern \(((t)-[E]->(h))\{0,3\}\) the result of this expansion would be

\[
\varepsilon \\
| (t) - [E] -> (h) \\
| (t) - [E] -> (h) (t') - [E'] -> (h') \\
| (t) - [E] -> (h) (t') - [E'] -> (h') (t'') - [E''] -> (h'')
\]

Any path that matched one of these four path conditions would satisfy the operand.

Our semantics extend straightforwardly to predicates on the pattern which is quantified. We provide examples of \(\alpha\)-expansions in Example 8, including showing how the in-pattern WHERE clause is handled.

---

\(^{13}\) We are not suggesting that quantification expansion to a possibly infinite (or extremely large, albeit bounded) set of union operators is done statically at compile time. We are merely using this “expand to a set of unions” device in order to express the semantics.
4.9.3. Semantic equivalence of *, +, and ?

- * - is semantically equivalent to \{0,\}
- + - is semantically equivalent to \{1,\}
- ? - is semantically equivalent to \{0,1\}

4.9.4. Semantics of the empty pattern, denoted by \(\varepsilon\)

The empty pattern (denoted by \(\varepsilon\)) matches nothing on its own, and does not affect other patterns when juxtaposed. The empty pattern arises from zero repetitions of a quantified pattern, and behaves as if no pattern had been written at all.

This means that:

- MATCH \(\varepsilon\) has no solutions and produces no results\(^{14}\)
- Given arbitrary patterns \(\beta\) and \(\gamma\):
  - \(\varepsilon \beta \Leftrightarrow \beta\)
  - \(\beta \varepsilon \Leftrightarrow \beta\)
  - \(\beta \varepsilon \gamma \Leftrightarrow \beta \gamma\)
  - \(\varepsilon|\beta \Leftrightarrow \beta\)\(^{15}\)
  - \(\varepsilon, \beta\) has no solutions and produces no results

To concretize the implications of \(\varepsilon\) on patterns, we distinguish between two types of quantified patterns:

- **Partial pattern**\(^{16}\): This is a pattern where the quantification is only expressed over the adjoining edges and intermediate vertices in a path; i.e. the source and target vertices (i.e. the two ultimate 'endpoint' vertices) are not included in the quantification.
  
  Examples include (quantification scope emboldened):
  
  (x:Person)((a)->(b))* (y) and (a)->*(b)

- **Full pattern**: This is a pattern where the quantification is expressed over the entire pattern. That is, the source and target vertices are included in the quantification.
  
  An example - with the quantification scope emboldened - would be ( (a) -> (b) ) *

\(^{14}\) Because of this, applying a quantification that includes zero repetitions on a whole pattern is equivalent to having that quantification start at 1 repetitions. It can thus be argued that quantifications on whole patterns should be required to start at 1 repetitions.

\(^{15}\) This does not mean that \(\varepsilon\) is insignificant in path pattern union, the distributivity of the surrounding patterns across the path pattern union (defined in Canonicalization rule 6 of Section 5.1) needs to be applied before \(\varepsilon\) can be removed.

\(^{16}\) We assign these names purely as a matter of expository convenience in this paper.
Evaluating $\varepsilon$ on a partial pattern would essentially mean that one is expressing a match on two juxtaposed vertex patterns (which must be matched to the same vertex in the graph). For example, MATCH (a) ->{0}(b) is equivalent to MATCH (a) (b). This extends to versions containing predicates:

- MATCH (x:Person)((a)->(b)){0}(y) ⇔ MATCH (x:Person)(y)
- MATCH (a)->{0}(b) WHERE a.foo=12 AND b.bar=19
  ⇔
  MATCH (a)(b) WHERE a.foo=12 AND b.bar=19

By contrast, evaluating $\varepsilon$ on a full pattern will always reduce to evaluating MATCH $\varepsilon$. One can therefore argue that, from a user’s point of view, the Kleene * operator on a full pattern makes no sense, and that the permitted lower bound ought always to be at least one; i.e the Kleene + operator\(^\text{17}\).

4.10. CHEAPEST and SHORTEST

Path patterns may express that only the shortest (by number of edges in the path) or cheapest (by the sum of the specified cost functions for each sub-pattern of the path pattern along the matched path) paths are to be matched by the pattern. At this stage, we believe that these features follow on straightforwardly from our proposed semantics without the need for special treatment at this point.

\(^{17}\) We note that a full pattern can always be transformed into a partial pattern by juxtaposing a source and target vertex pattern around the quantification. Thus, the full pattern ((a)->(b))\(^*\) becomes the partial pattern (x) ((a)->(b))\(^*\) (y)
5. Variable assignment and pattern canonicalization

The semantics of pattern matching is fundamentally based on the variables in the pattern. Reusing the same variables in multiple parts of the pattern is how Joins between patterns are expressed. For path pattern union, using the same variables between operands expresses that the patterns are to be union compatible (if they match the same path).

With the semantics of pattern matching based on the variables in the pattern, the semantics require a complete assignment of variables. All vertex patterns, edge patterns, and path patterns must have variables assigned for the semantic expression of what matches the pattern. Since we want to syntactically allow the user to leave out variables that are not of importance for what the query asks for, we introduce a notion of implicitly assigned variables (or, shorter, implicit variables) as well as a set of rules for how the implicit variables are assigned.

The implicit variable assignment rules are defined to satisfy the following goals:

- Assign a complete binding of variables (i.e. vertex, edge and path variables) such that the semantics of pattern matching can be defined in terms of relational operations.
  - This includes being able to differentiate between pattern operands of a path pattern union that are of different shape where the user has not included enough variables to differentiate between the projected results of these patterns.
- Allow the results of patterns of compatible shape to be combined in a path pattern union even if they do not have a complete set of explicitly assigned variables.\(^{19}\)
  - This implies that \((\) - \() \mid \) - \()) \) should be considered equivalent to \((\) - \().
- Allow the semantics to be independent of the order of the operands of path pattern union, in particular between label expressions and path macros.

The realisation of these goals is accomplished by the following general principles for implicit variable assignment:

1. Before implicit variables are assigned, the pattern is canonicalized to expose all parts of the pattern that should be named. See Section 5.1 for further details.
2. Implicit variables must not interfere with any explicit variables anywhere in the query.

\(^{18}\) These are indicated throughout the document in the style "\(_{some-variable-name}\)"

\(^{19}\) This unification process was conceived of as a solution to the notion brought up by Oskar van Rest in the 2019-06-25 meeting of the SQL-PG Ad-Hoc of DM32.2 in INCITS, where Oskar suggested that the user should be free to leave out variable names and still get set semantics.
3. When assigning implicit variables to the operands of a path pattern union, elements at the same position of structurally compatible operands must be assigned the same implicit variables (if no explicit variable has already been assigned). This is realized by assigning the same implicit variable to elements of patterns that are positionally compatible (see the box below).

   ○ For path variables, the same process applies for assigning variables positionally. However more work is needed on the algorithm for deciding where to introduce path variables. See Section 5.2 for an exploration of this. The utility of path variables is illustrated by Example 6.

4. Path macros are expanded with all their variables replaced with variables allocated according to the implicit variable assignment rules. This ensures that macro expansion is hygienic (i.e. that the variables of the macro do not interfere with the variables at the place where the macro is used) and provides the ability to define the semantics in such a way that the order between operands of path pattern union is independent of the contents of the operands. Section 7 (Path macros) provides further details.

Expanding on Principle number 3 above, the following definition underpins our proposed approach:

Consider two canonicalized path patterns, pp1 and pp2, each of which is an operand to the path pattern union operator in a pattern. An element pattern $e_1$ in pp1 is defined to be positionally compatible with an element pattern $e_2$ in pp2 if:

- Proceeding from the left of the pattern, and ignoring the directionality of the pattern (induced by the edge), $e_1$ occurs in precisely the same position in pp1 as $e_2$ in pp2
- This means that if $e_1$ is a vertex (resp. edge) pattern, then so is $e_2$

We note that it doesn't matter if the operands contain differing numbers of element patterns. For example, () - [] -> () and () -> () both have three elements (once canonicalized), whereas () - [] -> () - [] - () has five elements. Since these patterns are structurally incompatible, and would be assigned a different number of implicit variables, it is safe to assign implicit variables purely based on positional compatibility.

For example, the positionally compatible element patterns in the following path patterns are indicated by colour and number:

- $(e_1) - [e_2] -> (e_3)$
- $(e_1) <-[e_2] - (e_3)$
- $(e_1) - [e_2] - (e_3) - [e_4] -> (e_5)$

---

20 Two fixed patterns are structurally compatible if they contain the same number of element patterns (where juxtapositioned vertex patterns are considered to be the same vertex pattern). This is all we need to consider, since the canonicalization transforms everything into path pattern unions of fixed patterns.
5.1. Canonicalization of patterns

Patterns are canonicalized according to the following rules (iteratively until a fixed point is reached):

1. **Vertex patterns** are always canonical.
2. Short form **Edge patterns** are canonicalized to include a (empty) body.
   
   i.e. (e1) -> () is canonicalized to (e1) -> ()
3. **Path patterns** consisting of an **Edge pattern** without vertices are canonicalized by inserting empty vertices on each side of the **Edge pattern**.
4. **Quantified patterns** are canonicalized by ensuring that the quantified pattern is enclosed in parentheses, and by canonicalizing the quantifier to the \{\text{n,m}\} - and \{\text{n}\} -forms according to the semantic equivalences in Section 4.9. This ensures that all quantifications are converted into (potentially infinite) **Path pattern unions** of fixed patterns.
5. In **Juxtapositioned** patterns, remove any ε. Remaining juxtapositioned vertex patterns - where at least one of which lacks a vertex variable declaration - are consolidated into a single vertex pattern, retaining any vertex variable declaration and WHERE clause. If both vertex patterns have label expressions, they are combined using conjunction (\&).
6. Apply distributivity of **Path patterns** across **Path pattern union** (i.e. that \( \alpha (\beta | \gamma ) \delta \) is equivalent to \( \alpha \beta \delta | \alpha \gamma \delta \)) to “lift” the **Path pattern union** to the top level of the pattern.
7. Combine operands of **Path pattern union** that have the same variables in the same places.
   
   (This rule applies after implicit variables have been assigned.)
   
   Predicates (label expressions and WHERE) from the combined operands are combined by logical OR. Note that the predicates of each operand must be grouped together and combined as a whole with the predicates from the other operands. For example, the query
   
   \((:A) -> (:B) | (:C) -> (:D)\)
   
   should not be rewritten such that solutions to
   
   \((:A) -> (:D)\) or \((:C) -> (:B)\)
   
   are returned.

In summary, this canonicalizes all patterns to a single **Path pattern union** of fixed patterns.

The generation of a canonical form of an operand with a fixed-length pattern, or of the set of canonical forms required to represent a bounded variable-length pattern is very likely to be feasible at compile-time. The same is true for plausibly-estimated ‘upper limits’ of unbounded patterns arising from the use of a Kleene star. Statistics could be maintained (such as the length

---

21 We thank Fred Zemke of Oracle (who kindly reviewed a preliminary draft of this paper) for his suggested refinement of Rule 5.

22 This becomes clearer when considering all predicates (such as label expressions) in their canonical form - expressed in a “WHERE” clause.
of a particular path in the graph) along with details about the number of hops traversed in previous versions of the query - these could be used to 'hint' to the query engine that, say, 5 hops, rather than 500 hops, is being considered. In time, path-based indexes could also come into play to further inform optimization strategies. The intention of this paper is not to detail compile- or run-time strategies for evaluating such queries. However, we believe that run-time computation of the upper bounds to complete theoretically infinitely large sets of canonicalized operands at reasonable cost is likely to be feasible. We would therefore at this stage not wish to reject the possibility of supporting patterns containing Kleene star.

5.2. Path pattern variable assignment

The notion of being able to assign variables to a path within a parenthesized sub-pattern seems useful from a user perspective - for example, being able to express predicates over a path enclosed under a quantification. That provides us with the possibility to assign implicit variables for such paths. It would also be possible to break up patterns into sub components, and assign implicit path variables to those using some algorithm.

We can think of a few ways to identify sub-patterns to assign path variables to. We have however not yet analysed the options in deep enough detail to have an opinion on which method would seem most useful. The methods we have identified include:

1. Assign a single implicit path variable for each operand of the final path pattern union once the pattern has been fully canonicalized into a single path pattern union of fixed patterns.
2. Assign implicit path variables to each path pattern that the user has enclosed in parentheses.

There are benefits and drawbacks to both of these alternatives. A single path variable (1) seems overly simplistic. Implicit path variables for each parenthesis group (2) makes it harder to unify operands of a path pattern union.

Our intuition is that a simpler approach is better, so if it can be proven sufficient, it would seem appropriate to go with the simpler approach. The work of proving this is not yet completed, and it is possible that there are other options to consider as well.
6. Path pattern union

In this section, we present the semantics for the path pattern union operator.

Evaluation proceeds by first following the process of pattern canonicalization and implicit variable assignment as detailed in Section 5.

The evaluation of each operand of a Path pattern union results in a set of bindings (as described in Section 3, Background on identity and sets) - of elements to the variables - corresponding to the paths matching the operand.

We then perform a union between these sets, where this union is an “outer union” in that it will fill out the columns of each operand with nulls to match the combined columns of all operands. That means that if a variable is missing in an operand, then any solution row induced by the operand will have NULL assigned to the variable.

Thus, each set of results - induced by each operand - are combined by the path pattern union into a final set of results. The intuition behind this is that if the exact same set of variables - say, a, b and c - was used in more than one operand - say, op₁ and op₂ - and the same path p in the data was matched by both op₁ and op₂, then the binding of the elements in p to a, b and c will appear once in the final result set.

Operating under set semantics would mean that the path pattern union operation would be fully composable.

In summary, a single row is produced for every unique “matched path and variable-binding” combination.
6.1 Fixed patterns

We’ll begin by showing a few examples.

Assume we have the following graph, $G_1$:

![Figure 1. The graph $G_1$](image)

In the examples, we indicate the parts of interest in the patterns with **bold text**.

**Example 1: Positionally compatible elements with identical explicit variables**

The pattern

$$(a \text{ IS Animal})-[\mathbf{e}]->(b) \mid (a \text{ IS Cat})-[\mathbf{e}]->(b)$$

would produce the following rows (we’ve indicated which operand gave rise to each row):

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>e</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Left</strong></td>
<td>1</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td><strong>Left, Right</strong></td>
<td>2</td>
<td>22</td>
<td>4</td>
</tr>
</tbody>
</table>

Taking this step-by-step, the left operand gives rise to a set containing both rows, and the right operand gives rise to a set containing the second row only. The *path pattern union* operation combines both of these sets into the final set (shown above).

This shows that such a pattern could be rewritten to a label disjunction, as these two forms of patterns are equivalent:

$$(a \text{ IS Animal})-[\mathbf{e}]->(b) \mid (a \text{ IS Cat})-[\mathbf{e}]->(b) \iff (a \text{ IS Animal}|\text{Cat})-[\mathbf{e}]->(b)$$

However, caution needs to be exercised in generalizing this, and thus bears further investigation. For example, any such rewritings of a *path pattern union* to a simpler pattern...
p2 using label disjunctions must not lead to solutions (for p2) which would not be valid for p1. This is exemplified in **Example 2** and **Example 3** below.

Example 2: Positionally compatible elements with different explicit variables

The pattern

\[(a \text{ IS Animal})-[e]->(b) \mid (d \text{ IS Cat})-[e]->(b)\]

would produce the following rows (we’ve colour-coded them for visual ease, as well as indicated which operand gave rise to each row):

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>e</th>
<th>b</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>11</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>Left</td>
<td>2</td>
<td>22</td>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>22</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

In this case, we see the “outer union” semantics surfacing much more clearly: for the first operand, we see the same two rows as for **Example 1**, but now with the addition of a third row which is induced by the second operand. In effect, this means we have two rows for the same path p from vertex 2 to vertex 4 via edge 22 (essentially, multiset semantics insofar as the path p is concerned, although not where the variables are concerned\(^{23}\)).

Example 3: Positionally compatible elements, where one is implicit, and the other explicit

Hitherto, the examples have shown cases where the user explicitly assigned variables to all the patterns. From hereon in, we show our proposal for what happens when this is not the case.

In the case where a pattern in one operand has an explicit variable, but the positionally compatible pattern in the other operand does not, we propose that a differently-named variable is assigned implicitly. We illustrate this with the following example.

The pattern

\[(IS \text{ Animal})-[e]->(b) \mid (a \text{ IS Cat})-[e]->(b)\] //original pattern

would be rewritten to

\[^{23}\text{We recall that this is in line with our principle whereby a single row is produced for every unique “matched path and variable-binding” combination.}\]
\(_1 IS \text{Animal})-[e]->(b) \mid (a IS \text{Cat})-[e]->(b) \quad //\text{rewritten pattern}\)

The pattern would result in the following rows (we’ve colour-coded them for visual ease, as well as indicated which operand gave rise to each row):

<table>
<thead>
<tr>
<th>Operand</th>
<th>_1</th>
<th>a</th>
<th>e</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>null</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Left</td>
<td>2</td>
<td>null</td>
<td>22</td>
<td>4</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>2</td>
<td>22</td>
<td>4</td>
</tr>
</tbody>
</table>

At this point, it is reasonable to ask why the rewriting could not simply have assigned an implicit variable with the same name as the explicit variable to produce:

\((a IS \text{Animal})-[e]->(b) \mid (a IS \text{Cat})-[e]->(b)\)

After all, this would have resulted in the same results as Example 1, which at first glance may appear to make more sense, as one may reasonably suppose that the user simply forgot to provide an explicit variable for the first operand (and that in all likelihood this would have been intended to be the same variable).

However, upon further reflection, we realise that attempting to unite an explicit variable with an implicit one could actually lead to problems - even though it would solve the case laid out in the paragraph above. For example, assume the pattern (and that no vertex is ever both a Dog and a Cat):

\((\text{IS Dog})-[e]->(b) \mid (a IS \text{Cat})-[e]->(b)\)

In this case, the user may very well wish to express that any vertex bound to a must always be a Cat, and would expect the results to reflect this. Of course, the user expects to match on Dog vertices as well, but would in all likelihood not expect these to be bound to a, and if the results were to do this, it could lead to a lot of confusion (as to why a pattern that explicitly mandated binding to Cat vertices now also included bindings to Dog vertices).

Moreover, we also think it would be odd to conflate implicit and explicit variables in this way, particularly when it comes to projection. In other words, the user left the Dog vertices unnamed - for whatever reason - and to then bind them to the same variable as the explicitly-assigned one would cause more problems than it solves.
Example 4: Positionally compatible elements with identical implicit variables

In the case where two operands have no explicit variables in positionally compatible element patterns, we propose that the same implicit variable is assigned. We illustrate this with the following example.

The pattern (containing no explicit variables for the first vertex pattern in each operand)  

(\text{IS Animal})-[e]->(b) | (\text{IS Cat})-[e]->(b)

would, as a first step, be rewritten to  

(_1 \text{ IS Animal})-[e]->(b) | (_1 \text{ IS Cat})-[e]->(b)

This would produce the following rows (we note that only the last two columns would actually be projected):

<table>
<thead>
<tr>
<th>Operand</th>
<th>_1</th>
<th>e</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>Left, Right</td>
<td>2</td>
<td>22</td>
<td>4</td>
</tr>
</tbody>
</table>
Example 5: Achieving Multiset Alternation

This example demonstrates how path pattern union can be used to achieve Multiset Alternation. In other words, should the user desire the semantics of Multiset Alternation, there is a clear way in which to achieve this using our approach.

We use the same graph and pattern used in [JCJ-015]:

![Graph G2](image)

\[(p \text{ IS Person}) - [e] \rightarrow (q \text{ IS Person}) \mid (r) - [f \text{ IS Knows}] \rightarrow (s)\]

This would result in the following rows (we’ve colour-coded them for visual ease, as well as indicated which operand gave rise to each row):

<table>
<thead>
<tr>
<th>Operand</th>
<th>p</th>
<th>e</th>
<th>q</th>
<th>r</th>
<th>f</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>101</td>
<td>2</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Left</td>
<td>1</td>
<td>102</td>
<td>2</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>1</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>1</td>
<td>103</td>
<td>3</td>
</tr>
</tbody>
</table>

And, this is precisely the same set of rows as those mentioned in [JCJ-015]; in other words, complying with the semantics of Multiset Alternation.

In fact, we note that the same results could have been achieved by simply ensuring there is a difference of one variable. To illustrate, we’ll use the alternative formulation of the query, as given in [BNE-034]:
(a) ((IS Person) -> (IS Person) | -[IS Knows] -> ) (b)

Written in canonical form, this pattern becomes:

(a IS Person) -[]-> (b IS Person) | (a) -[IS Knows] -> (b)

After implicit variable assignment, this becomes:

(a IS Person) -[_2]-> (b IS Person) | (a) -[_2 IS Knows] -> (b)

This would result in the following rows:

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>_2</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left, Right</td>
<td>1</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>Left</td>
<td>1</td>
<td>102</td>
<td>2</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
<td>103</td>
<td>3</td>
</tr>
</tbody>
</table>

To emulate Multiset Alternation semantics with a change of one variable, the original query would simply need to have been written thus:

(a) ((IS Person) -[x] -> (IS Person) | -[y IS Knows] -> ) (b)

The evaluation of this query would produce:

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>x</th>
<th>y</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1</td>
<td>101</td>
<td>null</td>
<td>2</td>
</tr>
<tr>
<td>Left</td>
<td>1</td>
<td>102</td>
<td>null</td>
<td>2</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
<td>null</td>
<td>101</td>
<td>2</td>
</tr>
<tr>
<td>Right</td>
<td>1</td>
<td>null</td>
<td>103</td>
<td>3</td>
</tr>
</tbody>
</table>

This shows that our intention of being able to cater for both use cases - i.e. allowing for either multiset semantics or set semantics - using a single operator is indeed possible. All that is required is to ensure at least one unique explicit variable is assigned to a positionally compatible element pattern in the operands.

---

24 We note that this set of results would be achieved without actually evaluating each operand in turn, and then doing a set union over the two result sets (from each operand) by applying Canonicalization rule 7
Example 6: Considering path variables

Let’s now consider what happens when we bring in path variables.

Consider the following pattern:

\[(x) (\,(a) - \,[e]\, - \,(b) \mid (b) <- \,[e]\, - \,(a)\,) (y)\]

in which the user has provided the explicit variables a, e, and b. The variables x and y are provided as the start and end vertices of the matched path, to illustrate that the order in which a pattern is stated matters. The introduction of implicit path variables need to provide semantics that produce results consistent with the bindings for x and y, and do so even if x and y had not been present in the pattern.

Here, following the semantics of our proposal, we assign the same implicit path variable _1 to both operands.

\[(x) (_1 = (a) - [e] -> (b) \mid _1 = (b) <- [e] - (a)\,) (y)\]

We use the following graph, G₃:

![Figure 3. The graph G₃](image-url)
This would result in the following rows (first, we split out the rows for visual ease):

<table>
<thead>
<tr>
<th></th>
<th>(a) - [e] -&gt; (b)</th>
<th>(b) &lt;- [e] - (a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>_1 x</td>
<td>a e b y</td>
<td>_1 x a e b y</td>
</tr>
<tr>
<td>1, 11,5</td>
<td>1 11 5 5</td>
<td>5, 11, 1 5 1 11 5</td>
</tr>
<tr>
<td>2, 22,5</td>
<td>2 22 5 5</td>
<td>5, 22, 2 5 2 22 5 2</td>
</tr>
<tr>
<td>5, 55,5</td>
<td>5 55 5 5</td>
<td>5, 55, 5 5 5 55 5</td>
</tr>
</tbody>
</table>

We now show the result rows in a single table (we've colour-coded them for visual ease, as well as indicated which operand gave rise to which row):

<table>
<thead>
<tr>
<th>Operand</th>
<th>_1 x</th>
<th>a</th>
<th>e</th>
<th>b</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1, 11, 5</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>Left</td>
<td>2, 22, 5</td>
<td>2</td>
<td>2</td>
<td>22</td>
<td>5</td>
</tr>
<tr>
<td>Left, Right</td>
<td>5, 55, 5</td>
<td>5</td>
<td>5</td>
<td>55</td>
<td>5</td>
</tr>
<tr>
<td>Right</td>
<td>5, 11, 1</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>Right</td>
<td>5, 22, 2</td>
<td>5</td>
<td>2</td>
<td>22</td>
<td>5</td>
</tr>
</tbody>
</table>

We see that the assignment of the same implicit path variable, _1, correctly differentiates between the two operands and their stated order in the same way that the juxtaposition join with x and y does.

Projecting out the explicit variables a, e, and b would result in what seems like a multiset. However, this is not actually the case, as each operand results in a different path (bound to the implicit variable _1) with respect to the ordering of the elements therein; i.e. the path (1, 11, 5) is not the same as the path (5, 11, 1).

This is why there are no points at which patterns are rewritten to change the order from the order in which it was written. The order in which the user expressed the pattern is significant to the semantics of the pattern.

Aligning with the remarks made in [sql-pg-2019-0033] regarding the criticality of the sequential nature of paths, we note that the order of the elements within the path is very important.
We end this example by noting that had we applied the approach detailed in Example 5: Achieving Multiset Alternation as our default method for assigning implicit variables (here termed “M.A. Approach” for convenience), the pattern would have resulted in:

\[(x) \text{ (1) } = (a) - [e] --> (b) \mid \text{ (2) } = (b) <-- [e] - (a)) \text{ (y)}\]

This would result in the following rows (first split out in the same way as above):

<table>
<thead>
<tr>
<th>\text{(a) - [e] --&gt; (b)}</th>
<th>\text{(b) &lt;-- [e] - (a)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>_1 x a e b y</td>
<td>_2 x a e b y</td>
</tr>
<tr>
<td>1, 11, 5 1 1 11 5 5</td>
<td>5, 11, 1 5 1 11 5 1</td>
</tr>
<tr>
<td>2, 22, 5 2 2 22 5 5</td>
<td>5, 22, 2 5 2 22 5 2</td>
</tr>
<tr>
<td>5, 55, 5 5 5 55 5 5</td>
<td>5, 55, 5 5 5 55 5 5</td>
</tr>
</tbody>
</table>

Combined together, this produces the following results:

<table>
<thead>
<tr>
<th>Operand</th>
<th>_1</th>
<th>_2</th>
<th>x</th>
<th>a</th>
<th>e</th>
<th>b</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>1, 11, 5</td>
<td>null</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Left</td>
<td>2, 22, 5</td>
<td>null</td>
<td>2</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Left</td>
<td>5, 55, 5</td>
<td>null</td>
<td>5</td>
<td>5</td>
<td>55</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>5, 55, 5</td>
<td>5</td>
<td>5</td>
<td>55</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>5, 11, 1</td>
<td>5</td>
<td>1</td>
<td>11</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Right</td>
<td>null</td>
<td>5, 22, 2</td>
<td>5</td>
<td>2</td>
<td>22</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Table B

The difference between these results and the results given in Table A is the duplication of the row that matches the loop. Given which variables are explicitly projected from the query, this duplication adds no value, and argues in favour of our proposal. The outcome of the “M.A. Approach” is of course still achievable through explicit assignment of variables in the way that the “M.A Approach” would assign implicit variables, but our approach seems like a better one for the assignment of implicit variables.

---

25 Namely, the assignment of unique implicit variables for every pattern across the entire path pattern union.
Example 7: Operands of differing lengths

It is reasonable to consider how our approach works when faced with two operands of differing lengths. For example:

\[(a) \rightarrow (b) \mid (a) \rightarrow (\_2) \rightarrow (b)\]

Therefore, this will result in the following implicit variables being assigned to positionally compatible unnamed element patterns:

\[(a) \rightarrow [\_2] \rightarrow (b) \mid (a) \rightarrow [\_2] \rightarrow (\_3) \rightarrow [\_4] \rightarrow (b)\]

Although we’ve assigned the same implicit variables to all positionally compatible element patterns, this obviously results in a different number of variables in each operand. This means the operands are structurally incompatible (irrespective of whether or not they have the same implicit variables).

To see how this would work, let’s assume the following graph, \(G_4\):

![Figure 4. The graph \(G_4\)](image)

This would result in the following rows (we’ve colour-coded them for visual ease, as well as indicated which operand gave rise to each row):

<table>
<thead>
<tr>
<th>Operand</th>
<th>a</th>
<th>_2</th>
<th>_3</th>
<th>_4</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>v1</td>
<td>e1</td>
<td>null</td>
<td>null</td>
<td>v2</td>
</tr>
<tr>
<td>Left</td>
<td>v1</td>
<td>e3</td>
<td>null</td>
<td>null</td>
<td>v3</td>
</tr>
<tr>
<td>Left</td>
<td>v2</td>
<td>e2</td>
<td>null</td>
<td>null</td>
<td>v3</td>
</tr>
<tr>
<td>Right</td>
<td>v1</td>
<td>e1</td>
<td>v2</td>
<td>e2</td>
<td>v3</td>
</tr>
</tbody>
</table>
Projecting out the explicit variables (a and b) would result in a multiset for the **emboldened** rows (2 and 4). This is correct, as these two rows represent two different paths.
6.2 Quantified patterns

Continuing in the same vein as for fixed patterns, we now exemplify our approach with quantified patterns.

Example 8: Canonicalization of quantified patterns

In the following examples, we will explore a few queries and what their canonicalized form with implicit variables assigned would be, in order to illustrate various cases of canonicalization.

Further to the details specified in Semantics of the empty pattern, denoted by \( \epsilon \), we note that queries 1, 2, 5 and 6 are examples of partial patterns, and queries 3 and 4 are full patterns.

We will not be illustrating the result of these queries over a sample graph, as the previous examples provide sufficient material in that regard.

1. \( (a) \rightarrow^* (b) \mid (a) \rightarrow (b) \)

\( \Leftrightarrow (a) - [\] \rightarrow^* (b) \mid (a) - [\] \rightarrow (b) \)

according to Canonicalization rule 2

\( \Leftrightarrow (a) (() - [\] \rightarrow () )^* (b) \mid (a) - [\] \rightarrow (b) \)

according to Canonicalization rule 3

\( \Leftrightarrow (a) ( \epsilon \mid () - [\] \rightarrow () \mid () - [\] \rightarrow () \mid [\] - [\] \rightarrow () \mid \ldots ) (b) \mid (a) - [\] \rightarrow (b) \)

according to Canonicalization rule 4

\( \Leftrightarrow (a) \epsilon (b) \mid (a) () - [\] \rightarrow () (b) \mid (a) () - [\] \rightarrow () (b) \mid [\] - [\] \rightarrow () (b) \mid (a) \ldots (b) \mid (a) - [\] \rightarrow (b) \)

according to Canonicalization rule 6

\( \Leftrightarrow (a) (b) \mid (a) - [\] \rightarrow (b) \mid (a) - [\] \rightarrow () - [\] \rightarrow (b) \mid (a) \ldots (b) \mid (a) - [\] \rightarrow (b) \)

according to Canonicalization rule 5
The final result is equivalent to \((a)\rightarrow^* (b)\) since the other branch is redundant.

2. \((a)\ ( (x)\rightarrow(y))\ )^* (b)\ | (a)\rightarrow (b)

\[\Rightarrow (a)\ (b)\ | (a)\ (x)\rightarrow [\ ] \rightarrow (y)\ (b)\ | (a)\ (x)\rightarrow [\ ] \rightarrow (y)\ (x')\rightarrow [\ ] \rightarrow (y')\ (b)\ | (a)\ldots (b)\ | (a)\rightarrow [\ ] \rightarrow (b)\]

according to the Canonicalization rules

\[\Rightarrow _0=(a)\ (b)\ | _0=(a)\ (x)\rightarrow [\ ] \rightarrow (y)\ (b)\ | _0=(a)\ (x)\rightarrow [\ ] \rightarrow (y)\ (x')\rightarrow [\ ] \rightarrow (y')\ (b)\ | _0=(a)\ldots (b)\ | _0=(a)\rightarrow [\ ] \rightarrow (b)\]

by assignment of implicit variables

Here we see that the second operand of the original \emph{path pattern union} will \textbf{not} be eliminated, since it binds different variables from the one of equal length produced by the Kleene star.

3. \(( (x)\rightarrow(y)\ WHERE\ x.p < y.p )^* \ | ( (x)\rightarrow(y)\ WHERE\ x.p > y.p )\)

\[\Rightarrow \epsilon\ |\ ( (x)\rightarrow [\ ] \rightarrow (y)\ WHERE\ x.p < y.p )\ |\ ( (x)\rightarrow [\ ] \rightarrow (y)\ (x')\rightarrow [\ ] \rightarrow (y')\ WHERE\ x.p < y.p AND x'.p < y'.p )\ |\ ...\ |\ ( (x)\rightarrow [\ ] \rightarrow (y)\ WHERE\ x.p > y.p )\]

according to the Canonicalization rules
\[ _0 = \varepsilon \ |
\begin{align*}
( \ _0=(x)-\_2->(y) & \text{ WHERE } x.p < y.p ) \ | \\
( \ _0=(x)-\_2->(y) & (x')-\_4->(y') \\
\text{ WHERE } x.p < y.p \text{ AND } x'.p < y'.p ) \ | \\
\ldots \ | \\
( \ _0=(x)-\_2->(y) & \text{ WHERE } x.p > y.p )
\end{align*}
\]

by assignment of implicit variables

\[ _0 = \varepsilon \ |
\begin{align*}
( \ _0=(x)-\_2->(y) & \text{ WHERE } x.p < y.p \text{ OR } x.p > y.p ) \ | \\
( \ _0=(x)-\_2->(y) & (x')-\_4->(y') \\
\text{ WHERE } x.p < y.p \text{ AND } x'.p < y'.p ) \ | \\
\ldots 
\end{align*}

since the two patterns of equal length are compatible, we can combine the predicates.

(according to Canonicalization rule 7)

4. \(( x ) \rightarrow ( y ) \ast | ( n ) \rightarrow ( m )

\[ _0 = \varepsilon \ |
\begin{align*}
\begin{align*}
( x ) & \rightarrow ( y ) \ast \\
( n ) & \rightarrow ( m )
\end{align*}
\end{align*}
\]

according to the Canonicalization rules and assignment of implicit variables

At this point we see that the second operand of the original path pattern union will not be eliminated, since it binds different variables from the other pattern of equal length.

5. \(( a )-\[ e ]-( b ) | ( ( a )-\[ e ]-( b ) ( ( c )-\[ f ]-( d ) ) ) ? \)

\[ \begin{align*}
( a ) & -\[ e ]-( b ) \ | \ ( ( a )-\[ e ]-( b ) ( \varepsilon \ | \ ( c )-\[ f ]-( d ) ) ) \\
\text{according to Canonicalization rule 4}
\end{align*}
\]

\[ \begin{align*}
( a ) & -\[ e ]-( b ) \ | \ ( ( a )-\[ e ]-( b ) \varepsilon \ | \ ( a )-\[ e ]-( b ) ( c )-\[ f ]-( d ) ) \\
\text{according to Canonicalization rule 6}
\end{align*}
\]

\[ \begin{align*}
( a ) & -\[ e ]-( b ) \ | \ ( ( a )-\[ e ]-( b ) \ | \ ( a )-\[ e ]-( b ) ( c )-\[ f ]-( d ) ) \\
\text{according to Canonicalization rule 5}
\end{align*}
\]

\[ \begin{align*}
_0 & = ( a )-\[ e ]-( b ) \ | \ _0 = ( a )-\[ e ]-( b ) ( c )-\[ f ]-( d ) \\
\text{by assignment of implicit variables and elimination of equivalent operands}
\end{align*}
\]
6.  \( (a)-[:T]->*(:Person) \)
    | \( (a)-[:T]->() -[:T]->*:(:Person) \)

\[
\Leftrightarrow \ 
\begin{align*}
\ & (a)(()-[:T]->())*(:Person) \\
\ | \ & (a)-[:T]->() (()-[:T]->())*(:Person) \\
\ & (a)-[:T]->()(()-[:T]->())*(:Person) \\
\end{align*}
\]
according to **Canonicalization rule 3**

\[
\Leftrightarrow \ 
\begin{align*}
\ & (a)\varepsilon(:Person) \\
\ | \ & (a)()-[:T]->() (:Person) \\
\ & (a)()-[:T]->()()-[:T]->() (:Person) \\
\ & (a)...(:Person) \\
\ & (a)-[:T]->()\varepsilon(:Person) \\
\ | \ & (a)-[:T]->()()-[:T]->() (:Person) \\
\ & (a)-[:T]->()()-[:T]->()()-[:T]->() (:Person) \\
\ & (a)...(:Person) \\
\end{align*}
\]
according to **Canonicalization rule 4** and **Canonicalization rule 6**

\[
\Leftrightarrow \ 
\begin{align*}
\ & (a:Person) \\
\ & (a)-[:T]->(:Person) \\
\ & (a)-[:T]->()-[:T]->(:Person) \\
\ & (a)...(:Person) \\
\end{align*}
\]
according to **Canonicalization rule 5**

\[
\Leftrightarrow \ 
\begin{align*}
\ & (a:Person) \\
\ & (a)-[_2:T]->(_3:Person) \ // A \\
\ & (a)-[_2:T]->(_3) -[_4:T]->(_5:Person) \ // B \\
\ & (a)...(:Person) \ // C \\
\end{align*}
\]
by assignment of implicit variables and elimination of equivalent operands

*We note that the same pattern appears in both operands (highlighted in bold and named “A”, “B”, and “C” in comments), and that this will become equivalent to just \( (a)-[:T]->*(:Person) \), which is in line with the intuition for the initial expression of the query based on regular expression rewritings.*
7. Path macros

The semantics of pattern matching described in this paper naturally extends to *Path macros* as well. While *Path macros* is a topic worthy of a whole paper of its own, it is valuable to show how the *implicit variable assignment* interacts with *Path macros* and provides important and desirable traits to *Path macro* expansion and its interaction with *path pattern union*.

**Path macros are hygienic.** This means that the variables in the macro definition must not influence the variables in the pattern where the macro is expanded.

This is realized semantically by assigning new variables to all parts of the expanded macro, rewriting the references to the variables in the macro definition to use the newly assigned variables. This is realized by the *implicit variable assignment* algorithm described in Section 5. This is safe since the variables of the macro are not, in any case, accessible outside the macro definition. However, we might want to introduce ways of binding results computed by a *Path macro* to variables explicitly exposed by the macro, but that is a topic for another paper.

7.1. A basic example

Assume we have the macro `INHERITS` defined thus:

\[
\text{PATH} \; (x) -/\text{INHERITS} -/\rightarrow (y) = (x) - [A \mid B \mid C \mid D] -/\rightarrow (y)
\]

Now assume we have the pattern:

\[
(a) -/ :KNOWS \mid \text{INHERITS} \mid :LOVES -/\rightarrow (b)
\]

Expanding the macro results in the following pattern:

\[
(a) \ ( ( ) - [ :KNOWS ] -/\rightarrow ( ) \\
\mid (x) - [ A \mid B \mid C \mid D] -/\rightarrow (y) \\
\mid ( ) - [ :LOVES ] -/\rightarrow ( ) ) (b)
\]

Here, the underlined variables `x` and `y` need to be replaced according to the implicit variable assignment rules in order for the macro expansion to be hygienic. These same rules are at the same time used to assign all other missing variables, resulting in:

\[
(a) \ ( ( _e_1_s ) - [ _e_1 :KNOWS ] -/\rightarrow ( _e_1_t ) \\
\mid ( _e_1_s ) - [ _e_1 :A \mid B \mid C \mid D] -/\rightarrow ( _e_1_t ) \\
\mid ( _e_1_s ) - [ _e_1 :LOVES ] -/\rightarrow ( _e_1_t ) ) (b)
\]
Which is equivalent to:

```
(a) {
    (e1.s-[e1:KNOWS]->(e1.t) |
    (e1.s-[e1:A]->(e1.t) |
    (e1.s-[e1:B]->(e1.t) |
    (e1.s-[e1:C]->(e1.t) |
    (e1.s-[e1:D]->(e1.t) |
    (e1.s-[e1:LOVES]->(e1.t)
}
```

7.2. A more complex example

Given the following graph, \( G_{authors} \):

Assume we have the macro defined thus:

\[
\text{PATH CO_AUTHORED} = (x)-[:WROTE]->(y)<-[:WROTE]-(z)
\]

Now assume we have the pattern:

\[
(a)-/\text{CO_AUTHORED}/-(b)
\]

This is rewritten to:

\[
(a)-/\text{CO_AUTHORED}/->(b) | (a)<-/\text{CO_AUTHORED}/-(b)
\]
In an analogous manner to the previous example, we expand the macro, assign an implicit variable \_f to all instances of the path macro invocation, and then assign subscripted variations of \_f to the corresponding patterns within the expanded form of the path pattern union:

```
//operand 1
(a) ( (_f_1) - [ _f_2:WROTE] ->( _f_3) <- [ _f_4:WROTE] - ( _f_5) )  (b)

//operand 2
(a) ( (_f_1) - [ _f_2:WROTE] ->( _f_3) <- [ _f_4:WROTE] - ( _f_5) )  (b)
```

The following rows indicate the solutions to the pattern (the rows are split out for visual ease, and we indicate in (bold) the corresponding variable in the path macro definition):

<table>
<thead>
<tr>
<th>operand 1</th>
<th>operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>f_1</td>
</tr>
<tr>
<td>v1</td>
<td>v1</td>
</tr>
<tr>
<td>v3</td>
<td>v3</td>
</tr>
</tbody>
</table>

The first (resp. second) row is identical for both operands. In this instance, we would expect that during static analysis, the redundant union would be removed (so that only one of the operands is ever evaluated, not both), as illustrated in Example 8.

The final projected results from the original pattern (a) -/ CO_AUTHORED /-(b):

```
<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v3</td>
</tr>
<tr>
<td>v3</td>
<td>v1</td>
</tr>
</tbody>
</table>
```
9. An ITI and GQL community contribution from Neo4j Inc.

This contribution, if entered into an INCITS document register or repository, is a Deliverable under the terms of clause 2.2.1 of the Agreement for Membership in the InterNational Committee for Information Technology Standards ("INCITS"), a Division of the Information Technology Industry Council ("ITI") to which Neo4j Inc. is a party.

It is also a contribution to the GQL community and, like all such contributions, is:

Copyright © 2019 Neo4j Inc.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License.

You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Apache License, Version 2.0, Attribution Notice

This document is a contribution by Neo4j’s Query Languages Standards and Research group to the GQL community, and to the effort to create an ISO/IEC JTC 1 GQL graph query language project.