Querying RDF Databases with Sub-CONSTRUCTs
Dominique Duval, Rachid Echahed, Frédéric Prost

To cite this version:
Dominique Duval, Rachid Echahed, Frédéric Prost. Querying RDF Databases with Sub-CONSTRUCTs. 2021. hal-03268281

HAL Id: hal-03268281
https://hal.archives-ouvertes.fr/hal-03268281
Preprint submitted on 23 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Querying RDF Databases with Sub-CONSTRUCTs

Dominique Duval  
LJK - Univ. Grenoble Alpes  
dominique.duval@univ-grenoble-alpes.fr

Rachid Echahed  
LIG - Univ. Grenoble Alpes  
rachid.echahed@univ-grenoble-alpes.fr

Frédéric Prost  
LIG - Univ. Grenoble Alpes  
frederic.prost@univ-grenoble-alpes.fr

Graph query languages feature mainly two kinds of queries when applied to a graph database: those inspired by relational databases which return tables such as SELECT queries and those which return graphs such as CONSTRUCT queries in SPARQL. The latter are object of study in the present paper. For this purpose, a core graph query language GrAL is defined with focus on CONSTRUCT queries. Queries in GrAL form the final step of a recursive process involving so-called GrAL patterns. By evaluating a query over a graph one gets a graph, while by evaluating a pattern over a graph one gets a set of matches which involves both a graph and a table. CONSTRUCT queries are based on CONSTRUCT patterns, and sub-CONSTRUCT patterns come for free from the recursive definition of patterns. The semantics of GrAL is based on RDF graphs with a slight modification which consists in accepting isolated nodes. Such an extension of RDF graphs eases the definition of the evaluation semantics, which is mainly captured by a unique operation called Merge. Besides, we define aggregations as part of GrAL expressions, which leads to an original local processing of aggregations.

1 Introduction

Graph database query languages are becoming ubiquitous. In contrast to classical relational databases where SQL language is a standard, different languages [1] have been proposed for querying graph databases, like SPARQL [9] or Cypher [5]. Among the most popular models for representing graph databases, one may quote for instance the sets of triples (or RDF graphs [10]) used by SPARQL or the property graphs used by Cypher. In addition to the lack of a standard model to represent graph databases, there are different kinds of queries in the context of graph query languages. One may essentially distinguish two classes of queries: those inspired by relational databases which return tables such as SELECT queries and those which return graphs such as CONSTRUCT queries in SPARQL. Such CONSTRUCT queries are graph-to-graph queries specific to graph databases.

The graph-to-graph queries received less attention than the graph-to-table queries. For instance, for the SPARQL language, a semantics of SELECT queries and subqueries is proposed in [6], a semantics of CONSTRUCT queries in [7] and a semantics of CONSTRUCT queries with nested CONSTRUCT queries in FROM clauses in [2, 8], where the outcome of a subCONSTRUCT is a graph.

In this paper, we focus on graph-to-graph queries and subqueries for RDF graphs and we propose a new semantics for CONSTRUCT subqueries which departs from the one in [2, 8]. In fact, we define CONSTRUCT subpatterns rather than CONSTRUCT subqueries. For this purpose, we introduce a core query language GrAL based on RDF graphs. The syntactic categories of GrAL include both queries and patterns. When evaluating a CONSTRUCT query over a graph one gets a graph, whereas when evaluating a CONSTRUCT pattern over a graph one gets a set of matches which involves both variable
assignments and a graph. In fact, a CONSTRUCT query first acts as a CONSTRUCT pattern and then returns only the constructed graph. As the definition of patterns is recursive, CONSTRUCT subpatterns are obtained for free.

In order to define the semantics of GrAL, we introduce an algebra of operations over sets of matches, where a match is a morphism between graphs. We propose to base the semantics of GrAL upon an algebra on sets of matches, like the semantics of SQL is based upon relational algebra. All operations in our algebra essentially derive from a unique operation called merge, which generalizes the well-known join operation. We consider graphs consisting of classical RDF triples possibly augmented with some additional isolated nodes. This slight extension helps formulating the semantics of patterns and queries without using some cumbersome notations to handle, for instance, environments defined by variable bindings. The proposed algebra is used to define an evaluation semantics for GrAL. As for aggregations, they are handled locally inside expressions. The semantics of the various patterns and queries is uniform, as it is based on instances of the merge operation.

The paper is organized as follows. Section 2 introduces the algebra designed to express the semantics of queries in the Graph Algebraic Query Language GrAL. Graphs and matches are introduced in Section 2.1, then operations on sets of matches are defined in Section 2.2. In this paper, graphs are kinds of generalised RDF graphs that may contain isolated nodes. Let \( L \) be a set, called the set of labels, union of two disjoint sets \( C \) and \( V \), called respectively the set of constants and the set of variables.

**Definition 2.1** (graph). Every element \( t = (s, p, o) \) of \( L^3 \) is called a triple and its members \( s, p \) and \( o \) are called respectively the subject, predicate and object of \( t \). A graph \( X \) is made of a subset \( X_N \) of \( L \) called the set of nodes of \( X \) and a subset \( X_T \) of \( L^3 \) called the set of triples of \( X \), such that the subject and the object of each triple of \( X \) is a node of \( X \). The nodes of \( X \) which are neither a subject nor an object are called the isolated nodes of \( X \). The set of labels of a graph \( X \) is the subset \( L(X) \) of \( L \) made of the nodes and predicates of \( X \), then \( C(X) = C \cap L(X) \) and \( V(X) = V \cap L(X) \). Given two graphs \( X_1 \) and \( X_2 \), the graph \( X_1 \) is a subgraph of \( X_2 \), written \( X_1 \subseteq X_2 \), if \( (X_1)_N \subseteq (X_2)_N \) and \( (X_1)_T \subseteq (X_2)_T \), then obviously \( L(X_1) \subseteq L(X_2) \). The union \( X_1 \cup X_2 \) is the graph defined by \( (X_1 \cup X_2)_N = (X_1)_N \cup (X_2)_N \) and \( (X_1 \cup X_2)_T = (X_1)_T \cup (X_2)_T \). The intersection of two graphs without isolated nodes might have isolated nodes and \( L(X_1 \cap X_2) \) might be strictly smaller than \( L(X_1) \cap L(X_2) \), as for instance when \( X_1 = \{ (x, y, z) \} \) and \( X_2 = \{ (y, z, x) \} \) so that \( X_1 \cap X_2 = \{ x \} \).

**Definition 2.2** (match). A match \( m \) from a graph \( X \) to a graph \( G \), denoted \( m : X \rightarrow G \), is a function from \( L(X) \) to \( L(G) \) which preserves nodes and preserves triples and which fixes \( C \), in the sense that
of matches, all of them: (Definition 2.9 and 2.10). Then, these basic operations are combined in order to get some derived operations (Definition 2.11).

### 2.2 Operations on sets of matches

In this Section we introduce some operations on sets of matches which are used in Section 3 for defining the semantics of GrAL. The prominent one is the merging operation (Definition 2.8), which is a kind of generalized joining operation (see Definition 2.11). Other basic operations are the simple restriction and extension operations (Definitions 2.9 and 2.10). Then, these basic operations are combined in order to get some derived operations (Definition 2.11).
Definition 2.8 (Merge). Let $m : X \Rightarrow G$ be a set of matches and $p_m : Y \Rightarrow H_m$ a family of sets of matches indexed by $m \in m$, and let $H = \bigcup_{m \in m} H_m$. The merging of $m$ along the family $(p_m)_{m \in m}$ is the set of matches $m \triangleright p$ for every $m \in m$ and every $p \in p_m$ compatible with $m$:

$$\text{Merge}(m, (p_m)_{m \in m}) = \{ m \triangleright p \mid m \in m \land p \in p_m \land m \sim p \} : X \cup Y \Rightarrow G \cup H.$$ 

Let $q = \text{Merge}(m, (p_m)_{m \in m})$, then $q$ is made of a match $m \triangleright p$ for each pair $(m, p)$ with $m \in m$ and $p \in p_m$ compatible with $m$ (so that for each $m$ in $m$ the number of $m \triangleright p$ in $q$ is between 0 and $\text{Card}(p_m)$). The match $m \triangleright p : X \cup Y \Rightarrow G \cup H$ is such that $m \triangleright p(x) = m(x)$ when $x \in X$ and $m \triangleright p(y) = p(y)$ when $y \in Y$, which is unambiguous because of the compatibility condition.

Definition 2.9 (Restrict). Let $m : X \Rightarrow G$ be a set of matches. For every graph $Y$ contained in $X$ and every graph $H$ contained in $G$ such that $m(Y) \subseteq H$, the restriction $\text{Restrict}(m, Y, H) : Y \Rightarrow H$ is made of the restrictions of the matches in $m$ as matches from $Y$ to $H$. When $H = G$ the notation may be simplified:

$$\text{Restrict}(m, Y) = \text{Restrict}(m, Y, G) : Y \Rightarrow G.$$ 

Definition 2.10 (Extend). Let $m : X \Rightarrow G$ be a set of matches. For every graph $H$ containing $G$, the extension $\text{Extend}(m, H) : X \Rightarrow H$ is made of the extensions of the matches in $m$ as matches from $X$ to $H$.

New operations are obtained by combining the previous ones (assuming that $true$ is a constant). Comments on Definition 2.11 are given in Remark 2.12. We will see in Section 3.2 that these derived operations provide the semantics of the language GrAL. Examples are given in Section 4.

Definition 2.11 (derived operations).

- For every sets of matches $m : X \Rightarrow G$ and $p : Y \Rightarrow H$, let $p_m = p$ for each $m \in m$, then:
  $$\text{Join}(m, p) = \text{Merge}(m, (p_m)_{m \in m}) : X \cup Y \Rightarrow G \cup H.$$ 

- For every set of matches $m : X \Rightarrow G$, every family of constants $c = (c_m)_{m \in m}$ and every variable $x$, let $p_m = \{ p_m \}$ and $p_m(x) = c_m$ for each $m \in m$, then:
  $$\text{Bind}(m, c, x) = \text{Merge}(m, (p_m)_{m \in m}) : X \cup \{ x \} \Rightarrow G \cup c.$$ 

- For every set of matches $m : X \Rightarrow G$ and every family of constants $c = (c_m)_{m \in m}$, for some fresh variable $x$, let $true = (true)_{m \in m}$:
  $$\text{Filter}(m, c) = \text{Restrict}(\text{Bind}(m, c, true, x), X, G) : X \Rightarrow G.$$ 

- For every set of matches $m : X \Rightarrow G$ and every graph $R$, for every $m \in m$ let $p_m : R \Rightarrow p_m(R)$ be the match such that:
  $$p_m(x) = m(x) \text{ if } x \in \mathcal{V}(R) \cap \mathcal{V}(X)$$
  and $p_m(x) = \text{var}(x, m)$ is a fresh variable if $x \in \mathcal{V}(R) \setminus \mathcal{V}(X)$
  and let $p_m = \{ p_m \}$ and $p(R) = \bigcup_{m \in m} (p_m(R))$, then:
  $$\text{Construct}(m, R) = \text{Restrict}(\text{Merge}(m, (p_m)_{m \in m}), R) : R \Rightarrow G \cup p(R).$$ 

- For every sets of matches $m : X \Rightarrow G$ and $p : X \Rightarrow H$:
  $$\text{Union}(m, p) = \text{Extend}(m, G \cup H) \cup \text{Extend}(p, G \cup H) : X \Rightarrow G \cup H.$$ 

Remark 2.12. Let us analyse these definitions. Note that the definition of $\text{Bind}$ and $\text{Filter}$ rely on the fact that isolated nodes are allowed in graphs.

- Operation $\text{Join}$ is $\text{Merge}$ when the set of matches $p_m$ does not depend on $m$, so that:
  $$\text{Join}(m, p) = \{ m \triangleright p \mid m \in m \land p \in p \land m \sim p \} : X \cup Y \Rightarrow G \cup H.$$ 
  It follows that $\text{Join}$ is commutative.

- Operation $\text{Bind}$ is $\text{Merge}$ when $p_m$ has exactly one element $p_m$ for each $m$, which is such that $p_m(x) = c_m$. There are two cases:
The sets of matches obtained by the operations previously defined in this Section have bounded cardinalities, as follows.

\begin{align*}
\text{Card}(\text{Merge}(\underline{m}, (p_m)_{m \in \underline{m}})) & \leq \sum_{m \in \underline{m}}(\text{Card}(p_m)) \\
\text{Card}(\text{Restrict}(m, X, G)) & \leq \text{Card}(m) \\
\text{Card}(\text{Extend}(m, H)) & = \text{Card}(m) \\
\text{Card}(\text{Join}(m, p)) & \leq \text{Card}(m) \times \text{Card}(p) \\
\text{Card}(\text{Bind}(m, c, x)) & = \text{Card}(m) \\
\text{Card}(\text{Filter}(m, c)) & \leq \text{Card}(m) \\
\text{Card}(\text{Construct}(m, R)) & \leq \text{Card}(m) \\
\text{Card}(\text{Union}(m, p)) & \leq \text{Card}(m) + \text{Card}(p)
\end{align*}

The proof of Proposition 2.13 follows easily from the definitions.

## 3 The Graph Algebraic Query Language

In this Section we introduce the syntax and semantics of the Graph Algebraic Query Language GrAL. There are three syntactic categories in GrAL: expressions, patterns and queries. Expressions are considered in Section 3.1 Patterns are defined in Section 3.2 their semantics is presented as an evaluation
function which maps every pattern \( P \) and graph \( G \) to a set of matches \([P]_G\). Queries are defined in Section 3.3; they are essentially specific kinds of patterns and their semantics is easily derived from the semantics of patterns, the main difference is that the execution of a query on a graph returns simply a graph instead of a set of matches.

To each expression \( e \) or pattern \( P \) is associated a set of variables called its in-scope variables and denoted \( \mathcal{V}(e) \) or \( \mathcal{V}(P) \), respectively. An expression \( e \) is over a pattern \( P \) if \( \mathcal{V}(e) \subseteq \mathcal{V}(P) \). In this Section, as in Section 2, the set of labels \( \mathcal{L} \) is the union of the disjoint sets \( \mathcal{C} \) and \( \mathcal{V} \), of constants and variables respectively. We assume that the set \( \mathcal{C} \) of constants contains the numbers and strings and the boolean values true and false, as well as a symbol \( \bot \) for errors.

### 3.1 Expressions

The expressions of GrAL are built from the labels using operators, which are classified as either basic operators (unary or binary) and aggregation operators (always unary). Remember that typing constraints are not considered in this paper. Typically, and not exclusively, the sets \( Op_1, Op_2 \) and \( Agg \) of basic unary operators, basic binary operators and aggregation operators can be:

- \( Op_1 = \{ -, \text{NOT} \} \),
- \( Op_2 = \{ +, -, \times, /, \div, =, >, <, \text{AND}, \text{OR} \} \),
- \( Agg = \text{Agg}_{\text{elem}} \cup \{ \text{agg DISTINCT} \mid \text{agg} \in \text{Agg}_{\text{elem}} \} \),
  where \( \text{Agg}_{\text{elem}} = \{ \text{MAX, MIN, SUM, AVG, COUNT} \} \).

A group of expressions is a non-empty finite list of expressions.

**Definition 3.1** (syntax of expressions). The expressions \( e \) of GrAL and their set of in-scope variables \( \mathcal{V}(e) \) are defined recursively as follows:

- A constant \( c \in \mathcal{C} \) is an expression with \( \mathcal{V}(c) = \emptyset \).
- A variable \( x \in \mathcal{V} \) is an expression with \( \mathcal{V}(x) = \{ x \} \).
- If \( e_1 \) is an expression and \( op \in Op_1 \) then \( op e_1 \) is an expression with \( \mathcal{V}(op e_1) = \mathcal{V}(e_1) \).
- If \( e_1 \) and \( e_2 \) are expressions and \( op \in Op_2 \) then \( e_1 op e_2 \) is an expression with \( \mathcal{V}(e_1 op e_2) = \mathcal{V}(e_1) \cup \mathcal{V}(e_2) \).
- If \( e_1 \) is an expression and \( agg \in Agg \) then \( agg(e_1) \) is an expression with \( \mathcal{V}(agg(e_1)) = \mathcal{V}(e_1) \).
- If \( e_1 \) is an expression, \( agg \in Agg \) and \( gp \) a group of expressions with all its variables distinct from
  the variables in \( e_1 \), then \( agg(e_1 \text{ BY } gp) \) is an expression with \( \mathcal{V}(agg(e_1 \text{ BY } gp)) = \mathcal{V}(e_1) \).

The value of an expression with respect to a set of matches \( m \) (Definition 3.2) is a family of constants \( ev(m, e) = (ev(m, e)_m)_{m \in m} \) indexed by the set \( m \). Each constant \( ev(m, e)_m \) depends on \( e \) and \( m \) and it may also depend on other matches in \( m \) when \( e \) involves aggregation operators. The value of a group of expressions \( gp = (e_1, \ldots, e_k) \) with respect to \( m \) is the list \( ev(m, gp)_m = (ev(m, e_1)_m, \ldots, ev(m, e_k)_m) \). To each basic operator \( op \) is associated a function \( [op] \) (or simply \( op \)) from constants to constants if \( op \) is unary and from pairs of constants to constants if \( op \) is binary. To each aggregation operator \( agg \) in \( Agg \) is associated a function \( [(agg)] \) (or simply \( agg \)) from multisets of constants to constants. Note that each family of constants determines a multiset of constants: for instance a family \( \mathcal{C} = \{ c_m \}_{m \in m} \) of constants indexed by the elements of a set of matches \( m \) determines the multiset of constants \( \{ c_m \mid m \in m \} \), which is also denoted \( \mathcal{C} \) when there is no ambiguity. Some aggregation operators \( agg \) in \( Agg_{\text{elem}} \) are such that \( [(agg)](\mathcal{C}) \) depends only on the set underlying the multiset \( \mathcal{C} \), which means that \( [(agg)](\mathcal{C}) \) does not depend on the multiplicities in the multiset \( \mathcal{C} \): for instance this is the case for \( \text{MAX} \) and \( \text{MIN} \) but not for \( \text{SUM} \).
AVG, COUNT. When \( \text{agg} = \text{agg}_{\text{elem}} \) DISTINCT with \( \text{agg}_{\text{elem}} \) in \( \text{Agg}_{\text{elem}} \) then \([\text{agg}]\)(c) is \([\text{agg}_{\text{elem}}]\) applied to the underlying set of \( c \). For instance, COUNT \( c \) counts the number of elements of the multiset \( c \) with their multiplicities, while COUNT DISTINCT \( c \) counts the number of distinct elements in \( c \).

**Definition 3.2** (evaluation of expressions). Let \( X \) be a graph, \( e \) an expression over \( X \) and \( m : X \rightarrow Y \) a set of matches. The *value* of \( e \) with respect to \( m \) is the family of constants \( \text{ev}(m, e) = (\text{ev}(m, e)_m)_{m \in m} \) defined recursively as follows (with notations as in Definition 3.1):

- \( \text{ev}(m, c)_m = c \).
- \( \text{ev}(m, x)_m = m(x) \).
- \( \text{ev}(m, op e_1)_m = \text{ev}(m, e_1)_m \).
- \( \text{ev}(m, e_1 op e_2)_m = \text{ev}(m, e_1)_m \text{ev}(m, e_2)_m \).
- \( \text{ev}(m, \text{agg}(e_1))_m = \text{ev}(m, e_1)_m \) (which is the same for every \( m \) in \( m \)): see Example 4.2.
- \( \text{ev}(m, \text{agg}(e_1 \text{ BY gp } m'))_m = \text{ev}(m, e_1)_m \text{ev}(m, \text{gp})_m \) where \( m' \) in \( m \) such that \( \text{ev}(m, \text{gp})_m' = \text{ev}(m, \text{gp})_m \) (which is the same for every \( m \) and \( m' \) in \( m \) such that \( \text{ev}(m, \text{gp})_m' = \text{ev}(m, \text{gp})_m \)) see Example 4.3.

**Definition 3.3** (equivalence of expressions). Two expressions over a graph \( X \) are *equivalent* if they have the same value with respect to every set of matches \( m : X \rightarrow Y \).

### 3.2 Patterns

In Definition 3.4 the patterns of GrAL are built from graphs by using five operators: JOIN, BIND, FILTER, CONSTRUCT and UNION. In Definition 3.5 the semantics of patterns is given by an evaluation function. Some patterns have an associated graph called a **template** such a pattern \( P \) may give rise to a query \( Q \) as explained in Section 3.3, then the result of query \( Q \) is built from the template of \( P \).

**Definition 3.4** (syntax of patterns). The *patterns* \( P \) of GrAL, their set of *in-scope* variables \( \mathcal{V}(P) \) and their *template* graph \( \mathcal{T}(P) \) when it exists are defined recursively as follows.

- A graph is a pattern, called a *basic pattern*, and \( \mathcal{V}(P) \) is the set of variables of the graph \( P \).
- If \( P_1 \) and \( P_2 \) are patterns then \( P_1 \text{ JOIN } P_2 \) is a pattern and \( \mathcal{V}(P_1 \text{ JOIN } P_2) = \mathcal{V}(P_1) \cup \mathcal{V}(P_2) \).
- If \( P_1 \) is a pattern, \( e \) an expression over \( P_1 \) and \( x \) a variable then \( P_1 \text{ BIND } e \text{ AS } x \) is a pattern and \( \mathcal{V}(P_1 \text{ BIND } e \text{ AS } x) = \mathcal{V}(P_1) \cup \{x\} \).
- If \( P_1 \) is a pattern and \( e \) an expression over \( P_1 \) then \( P_1 \text{ FILTER } e \) is a pattern and \( \mathcal{V}(P_1 \text{ FILTER } e) = \mathcal{V}(P_1) \).
- If \( P_1 \) is a pattern and \( R \) a graph then \( P_1 \text{ CONSTRUCT } R \), also written CONSTRUCT \( R \) WHERE \( P_1 \) is a pattern and \( \mathcal{V}(P_1 \text{ CONSTRUCT } R) = \mathcal{V}(R) \).
  In addition this pattern has a template \( \mathcal{T}(P_1 \text{ CONSTRUCT } R) = R \).
- If \( P_1 \) and \( P_2 \) are patterns with template and if \( \mathcal{T}(P_1) = \mathcal{T}(P_2) \) then \( P_1 \text{ UNION } P_2 \) is a pattern and \( \mathcal{V}(P_1 \text{ UNION } P_2) = \mathcal{V}(R) \).
  In addition this pattern has a template \( \mathcal{T}(P_1 \text{ UNION } P_2) = \mathcal{T}(P_1) = \mathcal{T}(P_2) \).

The *value* of a pattern over a graph is a set of matches, as defined now.
Definition 3.5 (evaluation of patterns). The value of a pattern $P$ of GrAL over a graph $G$ is a set of matches $[[P]]_G : [P] \Rightarrow G^P$ from a graph $[P]$ that depends only on $P$ to a graph $G^P$ that contains $G$. This value $[[P]]_G : [P] \Rightarrow G^P$ is defined inductively as follows (with notations as in Definition 3.1):

- If $P$ is a basic pattern then $[[P]]_G = \text{Match}(P, G) : P \Rightarrow G$.
- $[[P_1 \text{ JOIN } P_2]]_G = \text{Join}([[P_1]]_G, [[P_2]]_G(G_{\text{const}})) : [P_1] \cup [P_2] \Rightarrow G^{P_1}(P_2)$.
- $[[P_1 \text{ BIND } e \text{ AS } x]]_G = \text{Bind}([[P_1]]_G, \text{ev}([[P_1]]_G, e), x) : [P_1] \cup \{x\} \Rightarrow G^{P_1} \cup [[P_1]]_G(e)$.
- $[[P_1 \text{ FILTER } e]]_G = \text{Filter}([[P_1]]_G, \text{ev}([[P_1]]_G, e)) : [P_1] \Rightarrow G^{P_1}$.
- $[[P_1 \text{ CONSTRUCT } R]]_G = \text{Construct}([[P_1]]_G, R) : R \Rightarrow G^{P_1} \cup [[P_1]]_G(R)$.
- $[[P_1 \text{ UNION } P_2]]_G = \text{Union}([[P_1]]_G, [[P_2]]_G(G_{\text{const}})) : R \Rightarrow G^{P_1}(P_2)$ where $R = \mathcal{T}(P_1) = \mathcal{T}(P_2)$.

Remark 3.6. Note that, syntactically, each operator OP builds a pattern $P$ from a pattern $P_1$ and a parameter param, which is either a pattern $P_2$ (for JOIN and UNION), a pair $(e, x)$ made of an expression $e$ and a variable (for BIND), an expression $e$ (for FILTER) or a graph $R$ (for CONSTRUCT). Semantically, for every non-basic pattern $P = P_1 \text{ OP param}$, we denote $m_1 : X_1 \Rightarrow G_1$ for $[[P_1]]_G : [P_1] \Rightarrow G^{P_1}$ and $m : X \Rightarrow G'$ for $[[P]]_G : [P] \Rightarrow G^P$. In every case it is necessary to evaluate $m_1$ before evaluating $m$: for JOIN and UNION this is because pattern $P_2$ is evaluated on $G_1$, for BIND and FILTER because expression $e$ is evaluated with respect to $m_1$, and for CONSTRUCT because of the definition of Construct. According to Definition 3.5 given a pattern $P$ and a graph $G$, the value $m : X \Rightarrow G'$ of $P$ is determined as follows:

- When $P$ is a basic pattern then $X = P$, $G' = G$ and $m$ is made of all matches from $P$ to $G$ (Example 4.1).
- $P = P_1 \text{ OP param}$ then the semantics of $P$ is easily derived from Definition 2.11 (see also Remark 2.12). However, note that the semantics of $P_1 \text{ JOIN } P_2$ and $P_1 \text{ UNION } P_2$ is not symmetric in $P_1$ and $P_2$ in general, unless $G^{P_1} = G$ (Examples 4.5 and 4.6).

Given a non-basic pattern $P = P_1 \text{ OP param}$, the pattern $P_1$ is a subpattern of $P$, as well as $P_2$ when $P = P_1 \text{ JOIN } P_2$ or $P = P_1 \text{ UNION } P_2$. The semantics of patterns is defined in terms of the semantics of its subpatterns (and the semantics of its other arguments, if any). Thus, for instance, CONSTRUCT patterns can be nested at any depth (Examples 4.4 and 4.5).

Proposition 3.7. For every pattern $P$, the set $\mathcal{V}(P)$ of in-scope variables of $P$ is the same as the set $\mathcal{V}(\mathcal{T}(P))$ of variables of the graph $[P]$.

Definition 3.8 (equivalence of patterns). Two patterns are equivalent if they have the same value over $G$ for every graph $G$.

Proposition 3.9. For every basic patterns $P_1$ and $P_2$, the basic pattern $P_1 \cup P_2$ is equivalent to $P_1 \text{ JOIN } P_2$ and to $P_2 \text{ JOIN } P_1$.

3.3 Queries

A query in GrAL is essentially a pattern which has a template. The main difference between patterns and queries is that, while a pattern is interpreted as a function from graphs to sets of matches, a query is interpreted as a function from graphs to graphs. The operator for building queries from patterns is denoted \text{GRAPH}. According to Definition 3.5, the value of a pattern $P$ with template $R$ over a graph $G$ is a set of matches $[[P]]_G : R \Rightarrow G^P$, and the semantics of patterns is defined recursively in terms of their values. Thus, patterns have a graph-to-set-of-matches semantics, while queries have a graph-to-graph semantics, as defined below, based on Definition 2.6 of the image of a graph by a set of functions.
The meaning of D. Duval, R. Echahed & F. Prost stamped at dates date1
date2.

The pattern of Q is P and the template \( \mathcal{T}(Q) \) of Q is the template of P.

**Definition 3.11** (result of queries). The result of a query Q with pattern P and template R over a graph G is the subgraph of G\((P)\) image of R by \([[[P]]]_G\), it is denoted Result\((Q,G)\).

Thus, when \( Q = \text{GRAPH} (P_1 \text{ CONSTRUCT} R) \), the result of Q over G is the graph Result\((Q,G) = [[[P_1]]]_G(R)\) built by “gluing” the graphs \(m(R)\) for \(m \in [[[P_1]]]_G\), where \(m(R)\) is a copy of R with each variable \(x \in \mathcal{V}(R) \setminus \mathcal{V}(X)\) replaced by a fresh variable \(\text{var}(x,m)\). And when \( Q = \text{GRAPH} (P_1 \text{ UNION} P_2) \), the result of Q over G is the graph Result\((Q,G) = H_1 \cup H_2\) where \(H_i = \text{Result}(\text{GRAPH}(P_i),G)\) and the fresh variables occuring in \(H_1\) are distinct from the ones in \(H_2\).

**Definition 3.12** (equivalence of queries). Two queries are equivalent if they have the same template and the same result over every graph.

It follows that queries with equivalent patterns are equivalent, but this condition is not necessary.

**Remark 3.13** (about SPARQL queries). CONSTRUCT queries in SPARQL are similar to CONSTRUCT queries in GrAL: the variables in \(\mathcal{V}(R) \setminus \mathcal{V}(X)\) in GrAL play the same role as the blank nodes in SPARQL. However the subCONSTRUCT patterns are specific to GrAL. There is no SELECT query in this core version of GrAL, however following [4] we may consider SELECT queries as kinds of CONSTRUCT queries.

### 4 Examples

In this Section we illustrate our concepts on a toy database that is a simplified view of a social media network. The network consists in authors publishing messages. Each message is timestamped at a certain date (a day). A message can refer to other messages and an author may like a message. An instance of such a network is described by the following graph G (written “à la” RDF):

```plaintext
auth1 publishes mes1 . auth1 publishes mes2 .
auth2 publishes mes3 . auth3 publishes mes4 . auth3 publishes mes5 .
mes1 stampedAt date1 . mes2 stampedAt date2 .
mes3 stampedAt date1 . mes4 stampedAt date4 . mes5 stampedAt date4 .
mes3 refersTo mes1 . mes4 refersTo mes1 . mes4 refersTo mes2 .
auth1 likes mes3 . auth1 likes mes4 . auth1 likes mes5 .
auth2 likes mes1 . auth2 likes mes4
```

The meaning of G is that author auth1 has published messages mes1 and mes2, which have been stamped respectively at dates date1 and date2, etc. Now we illustrate the evaluation of some GrAL queries of shape \( Q_i = \text{GRAPH}(P_i) \) where the pattern \(P_i\) is written in a SPARQL-like syntax. With the exception of Example [4.6] the pattern \(P_i\) has shape \(P_i = P'_i \text{ CONSTRUCT} R_i\), which is written as \(P_i = \text{CONSTRUCT} R_i\) WHERE \(P'_i\) for some template \(R_i\) and some pattern \(P'_i\). Thus, we know that the result of \(Q_i\) applied to graph G is an instance of \(R_i\) when \(\mathcal{V}(R_i) \subseteq \mathcal{V}(P'_i)\), and that in general it is built by “gluing” together several instances of \(R_i\) (as in Example [4.7]). In the following examples, the value of a pattern \(P\) over G is given by its graph \([P]\) and the assignment table \(\text{Tab}([[P]])_G\). The graph \(G[[P]]\) can be easily computed.
4.1 Basic pattern: author citations

In this example, the goal is to build the graph of author citations, where an author \( a_1 \) cites an author \( a_2 \) if \( a_1 \) has published a message that refers to a message published by \( a_2 \). In this example \( P'_1 \) is a basic pattern, i.e., it is a graph, and pattern \( P_1 \) is written as follows:

\[
\text{CONSTRUCT} \{ \ ?a_1 \text{ cite } \ ?a_2 \ \} \\
\text{WHERE} \{ \ ?a_1 \text{ publishes } \ ?m_1 \ . \ ?m_1 \text{ refersTo } \ ?m_2 \ . \ ?a_2 \text{ publishes } \ ?m_2 \ \}
\]

The result of query \( Q_1 = \text{GRAPH} (P_1) \) over \( G \) is the graph:

\[
\text{Result}(Q_1, G) = \{ \ \text{auth2 cites auth1} . \ \text{auth3 cites auth1} \}.
\]

4.2 Aggregation: number of likes

The goal here is to count the number of likes in the database. The result of query \( Q_2 \) is a number, which is considered in GrAL as a graph made of only one isolated node. We have to count the number of triples with predicate \( \text{likes} \), or equivalently the number of predicates \( \text{likes} \). Here is the pattern \( P_2 \):

\[
\text{CONSTRUCT} \{ \ ?n \ \} \\
\text{WHERE} \{ \ ?a \text{ likes } \ ?m \ \}
\text{BIND COUNT(likes) AS } \ ?n \ \}
\]

The result of \( Q_2 \) over \( G \) is the graph:

\[
\text{Result}(Q_2, G) = \{ 5 \}.
\]

Note that we would get a query equivalent to \( Q_2 \) by counting either the number of authors \( ?a \) who like a message (with multiplicity the number of messages liked by \( ?a \)), or by counting the number of messages \( ?m \) which are liked by someone (with multiplicity the number of authors who like \( ?m \)). This means that the line \( \text{BIND COUNT(likes) AS } ?n \) could be replaced by \( \text{BIND COUNT(?a) AS } ?n \) or by \( \text{BIND COUNT(?m) AS } ?n \).

However these two variants are less close to the goal, which may be error-prone.

4.3 Aggregation by group: number of likes per author

The goal in this example is to compute the number of likes per author. We display the result as the graph made of the triples \( ?a \text{ nb_of_likes } ?n \) where \( ?n \) is the number of likes of messages published by author \( ?a \), except for self-likes.

\[
\text{CONSTRUCT} \{ \ ?a \text{ nb_of_likes } ?n \ \} \\
\text{WHERE} \{ \ ?a \text{ publishes } ?m \ . \ ?a_1 \text{ likes } ?m \ \}
\text{FILTER (NOT(a1=a))} \\
\text{BIND COUNT(likes BY } ?a) \text{ AS } ?n \ \}
\]

The result of \( Q_3 \) over \( G \) is the graph:

\[
\text{Result}(Q_3, G) = \{ \ \text{auth1 nb_of_likes 1} . \ \text{auth2 nb_of_likes 1} . \ \text{auth3 nb_of_likes 3} \}.
\]

4.4 Subpattern: number of friends per author

In this query, the goal is to count the number of friends of each author, where friendship is the symmetric relation between authors defined as follows: two authors are \( \text{friends} \) when each one likes a publication by the other (here self-friends are allowed, otherwise a FILTER has to be added as in Example 4.3). We display the result as the graph made of the triples \( ?a \text{ nb_of_friends } ?n \) where \( ?n \) is the number of friends of author \( ?a \). Here we use a subpattern for building the graph of friendship.
CONSTRUCT { ?a1 nb_of_friends ?n }
WHERE {
  CONSTRUCT { ?a1 friend ?a2 }
  WHERE {
    ?a1 publishes ?m1 . ?a2 likes ?m1 . ?a2 publishes ?m2 . ?a1 likes ?m2 }
  BIND COUNT (friend BY ?a1) AS ?n }

The result of query $Q_4$ over $G$ is:
\[
\text{Result}(Q_4, G) = \{ \text{auth1 nb_of_friends 1, auth2 nb_of_friends 1} \}.
\]

4.5 Join: friendship relations

The subpattern $P'_4,1$ in Example 4.4 builds the graph of friendship. Here is another pattern $P_5$ for the same purpose, which uses the symmetry of the friendship relation:

CONSTRUCT { ?a1 friend ?a2 }
WHERE {
  CONSTRUCT { ?a1 friend ?a2 } WHERE { ?a1 publishes ?m . ?a2 likes ?m }
  JOIN { ?a2 friend ?a1 } }

The result of query $Q_5$ is the friendship relation:
\[
\text{Result}(Q_5, G) = \{ \text{auth1 friend auth2, auth2 friend auth1} \}.
\]

4.6 Union

The next query $Q_6$ builds the graph of relation $foo$, defined as follows: two authors are related by $foo$ when at least one of them likes a publication by the other.

CONSTRUCT { ?a1 foo ?a2 } WHERE { ?a1 publishes ?m1 . ?a2 likes ?m1 }
UNION
CONSTRUCT { ?a1 foo ?a2 } WHERE { ?a2 foo ?a1 }

The result of query $Q_6$ is:
\[
\text{Result}(Q_6, G) = \{ \text{auth1 foo auth2, auth2 foo auth1, auth1 foo auth3, auth3 foo auth1, auth2 foo auth3, auth3 foo auth2} \}.
\]

4.7 Fresh variables

Here the goal is to build, for each author $?a$ and each message $?m$ published by $?a$ and stamped at date $?d$, a tree with a fresh variable as root and with two branches, one named author towards $?a$ and the other one named date towards $?d$. The pattern $P_7$ is:

CONSTRUCT { ?r author ?a . ?r date ?d }
WHERE { ?a publishes ?m . ?m stampedAt ?d }

Note that the variable $?r$ in $R_7$ does not occur in $P'_7$. In fact, the query $Q_7$ “mimicks” the following SELECT query $Q'_7$:

SELECT ?a ?d WHERE { ?a publishes ?m . ?m stampedAt ?d }

As explained in [4], the various copies of the variable $?r$ in the result of $Q_7$ act as identifiers for the rows in the table result of $Q'_7$ over $G$ (as for instance in SPARQL), which is obtained by dropping the column $?r$ from $\text{Tab}([[P_7]])_G$.
5 Conclusion

We considered the problem of the evaluation of graph-to-graph queries, namely CONSTRUCT queries, possibly involving nested sub-queries. We proposed a new evaluation semantics of such queries which rests on a uniform definition of the notion of patterns. The evaluation of a pattern always yields a pair consisting of a graph and a set of matches (variable assignments). Notice that we did not tackle explicitly graph-to-table queries such as the well-known SELECT queries. We have shown recently in [4] that SELECT queries are particular case of CONSTRUCT queries. This stems from an easy encoding of tables as graphs. Thus, the present work can be extended immediately to SELECT queries involving Sub-SELECT queries.

The present work opens several perspectives including a generalization of the proposed semantics to other models of graphs such as property graphs. Such an extension needs to ensure the existence of the main operations of the proposed algebra such as the Merge operation. An operational semantics, based on rewriting systems, which is faithful with the evaluation semantics proposed in this paper is under progress. Its underlying rewrite rules are inspired by the algebraic approach in [3]. Furthermore, the core language GrAL contains only simple patterns needed to illustrate our uniform semantics. Comparison with other patterns such as FROM(query) [2, 8] or expressions such as EXISTS(pattern) remains to be investigated.

References


A Examples: details

In this Appendix we illustrate our concepts on a toy database that is a simplified view of a social media network. The network consists in authors publishing messages. Each message is timestamped at a certain date (a day). A message can refer to other messages and an author may like a message. An instance of such a network is described by the following graph $G$ (written “à la” RDF):

```
auth1 publishes mes1 . auth1 publishes mes2 .
auth2 publishes mes3 . auth3 publishes mes4 . auth3 publishes mes5 .
mes1 stampedAt date1 . mes2 stampedAt date2 .
mes3 stampedAt date1 . mes4 stampedAt date4 . mes5 stampedAt date4 .
mes3 refersTo mes1 . mes4 refersTo mes1 . mes4 refersTo mes2 .
auth1 likes mes3 . auth1 likes mes4 . auth1 likes mes5 .
auth2 likes mes3 . auth2 likes mes4
```

The meaning of $G$ is that author auth1 has published messages mes1 and mes2, which have been stamped respectively at dates date1 and date2, etc. Now we illustrate the evaluation of some GrAL queries of shape $Q_i = \text{GRAPH}(P_i)$ where the pattern $P_i$ is written in a SPARQL-like syntax. With the exception of Example A.6, the pattern $P_i$ has shape $P_i = P'_i \text{ CONSTRUCT } R_i$, which is written as $P_i = \text{CONSTRUCT } R_i \text{ WHERE } P'_i$ for some template $R_i$ and some pattern $P'_i$. Thus, we know that the result of $Q_i$ applied to graph $G$ is an instance of $R_i$ when $\mathcal{V}(R_i) \subseteq \mathcal{V}(P'_i)$, and that in general it is built by “gluing” together several instances of $R_i$ (as in Example A.7). In the following examples, the value of a pattern $P$ over $G$ is given by its graph $[P]$ and the assignment table $\text{Tab}([P]_G)$. The graph $G(P)$ can be easily computed.

A.1 Basic pattern: author citations

In this example, the goal is to build the graph of author citations, where an author $a_1$ cites an author $a_2$ if $a_1$ has published a message that refers to a message published by $a_2$. In this example $P'_i$ is a basic pattern, i.e., it is a graph, and pattern $P_i$ is written as follows:

```
CONSTRUCT { ?a1 cite ?a2 }
WHERE { ?a1 publishes ?m1 . ?m1 refersTo ?m2 . ?a2 publishes ?m2 }
```

The value of pattern $P'_i = \{ ?a1 publishes ?m1 . ?m1 refersTo ?m2 . ?a2 publishes ?m2 \}$ over $G$ is the set of matches $[[P'_i]]_G : P'_i \Rightarrow G$ with assignment table (Definition 2.5):

```
<table>
<thead>
<tr>
<th>?a1</th>
<th>?m1</th>
<th>?m2</th>
<th>?a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth2</td>
<td>mes3</td>
<td>mes1</td>
<td>auth1</td>
</tr>
<tr>
<td>auth3</td>
<td>mes4</td>
<td>mes1</td>
<td>auth1</td>
</tr>
<tr>
<td>auth3</td>
<td>mes4</td>
<td>mes2</td>
<td>auth1</td>
</tr>
</tbody>
</table>
```

Since pattern $P_i$ is $\text{CONSTRUCT } R_i \text{ WHERE } P'_i$ with $R_i = \{ ?a1 cites ?a2 \}$, so that $\mathcal{V}(R_i) \subseteq \mathcal{V}(P'_i)$, the value of pattern $P_i$ over $G$ is the set of matches $[[P_i]]_G : R_i \Rightarrow G$ with assignment table:

```
<table>
<thead>
<tr>
<th>?a1</th>
<th>?a2</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth2</td>
<td>auth1</td>
</tr>
<tr>
<td>auth3</td>
<td>auth1</td>
</tr>
</tbody>
</table>
```

Finally the result of query $Q_1 = \text{GRAPH}(P_1)$ over $G$ is the graph:

$\text{Result}(Q_1, G) = \{ \text{auth2 cites auth1 . auth3 cites auth1 } \}$
A.2 Aggregation: number of likes

The goal here is to count the number of likes in the database. The result of query $Q_2$ is a number, which is considered in GrAL as a graph made of only one isolated node. We have to count the number of triples with predicate `likes`, or equivalently the number of predicates `likes`. Here is the pattern $P_2$:

\[
\text{CONSTRUCT \{ ?n \}}
\]

\[
\text{WHERE \{ }
\]

\[
\text{ ?a likes ?m}
\]

\[
\quad \text{BIND COUNT(likes) AS ?n}
\]

\[
\text{\}}
\]

The template $R_2$ is a graph made of only one isolated node which is a variable `?n`. The graph $[P_2]$ is ` {?a likes ?m . ?n}` and the assignment table of $[[P_2]]_G$ is:

<table>
<thead>
<tr>
<th>?a</th>
<th>?m</th>
<th>?n</th>
</tr>
</thead>
<tbody>
<tr>
<td>auth1</td>
<td>mes3</td>
<td>5</td>
</tr>
<tr>
<td>auth1</td>
<td>mes4</td>
<td>5</td>
</tr>
<tr>
<td>auth1</td>
<td>mes5</td>
<td>5</td>
</tr>
<tr>
<td>auth2</td>
<td>mes1</td>
<td>5</td>
</tr>
<tr>
<td>auth2</td>
<td>mes4</td>
<td>5</td>
</tr>
</tbody>
</table>

Thus the result of $Q_2$ over $G$ is the graph:

\[
\text{Result}(Q_2, G) = \{ 5 \}
\]

Note that we would get a query equivalent to $Q_2$ by counting either the number of authors `?a` who like a message (with multiplicity the number of messages liked by `?a`), or by counting the number of messages `?m` which are liked by someone (with multiplicity the number of authors who like `?m`). This means that the line `BIND COUNT(likes) AS ?n` could be replaced by `BIND COUNT(?a) AS ?n` or by `BIND COUNT(?m) AS ?n`.

However these two variants are less close to the goal, which may be error-prone.

A.3 Aggregation by group: number of likes per author

The goal in this example is to compute the number of likes per author. We display the result as the graph made of the triples `?a nb_of_likes ?n` where `?n` is the number of likes of messages published by author `?a`, except for self-likes.

\[
\text{CONSTRUCT \{ ?a nb_of_likes ?n \}}
\]

\[
\text{WHERE \{ }
\]

\[
\text{ ?a publishes ?m . }
\]

\[
\text{ ?a1 likes ?m}
\]

\[
\quad \text{FILTER (NOT(a1=a))}
\]

\[
\quad \text{BIND COUNT(likes BY ?a) AS ?n}
\]

\[
\text{\}}
\]
The graph \([P_3']\) is \{\(?a\ \text{publishes}\ ?m, \ ?a1\ \text{likes}\ ?m, \ ?n\)\} and:

\[
\begin{array}{|c|c|c|}
\hline
?a & ?m & ?a1 \\
\hline
\text{auth1} & \text{mes1} & \text{auth2} \\
\text{auth2} & \text{mes3} & \text{auth1} \\
\text{auth3} & \text{mes4} & \text{auth1} \\
\text{auth3} & \text{mes5} & \text{auth1} \\
\hline
\end{array}
\]

Then the graph \([P_3]\) is \{\(?a\ \text{nb\_of\_likes}\ ?n\)\} and:

\[
\begin{array}{|c|c|}
\hline
?a & ?n \\
\hline
\text{auth1} & 1 \\
\text{auth2} & 1 \\
\text{auth3} & 3 \\
\hline
\end{array}
\]

so that the result of \(Q_3\) over \(G\) is the graph:

\[
\text{Result}(Q_3, G) = \{\text{auth1 nb\_of\_likes 1, auth2 nb\_of\_likes 1, auth3 nb\_of\_likes 3}\}
\]

### A.4 Subpattern: number of friends per author

In this query, the goal is to count the number of friends of each author, where friendship is the symmetric relation between authors defined as follows: two authors are \textit{friends} when each one likes a publication by the other (here self-friends are allowed, otherwise a \texttt{FILTER} has to be added as in Example A.3).

We display the result as the graph made of the triples \(?a\ \text{nb\_of\_friends}\ ?n\) where \(?n\) is the number of friends of author \(?a\). Here we use a subpattern for building the graph of friendship.

```r
CONSTRUCT \{ ?a1 \text{nb\_of\_friends} ?n \} 
WHERE 
{ 
  CONSTRUCT \{ ?a1 \text{friend} ?a2 \} 
  WHERE 
  { 
    ?a1 \text{publishes} ?m1 . ?a2 \text{likes} ?m1 . 
    ?a2 \text{publishes} ?m2 . ?a1 \text{likes} ?m2 
  } 
  BIND \text{COUNT} (\text{friend BY} \ ?a1) \ AS \ ?n 
}
```

Syntactically here \(Q_4 = \text{GRAPH} (P_4)\) with \(P_4 = P_4'\ \text{CONSTRUCT} R_4\) and \(P_4' = P_4'_{1}\ \text{BIND} e_4 \ AS \ ?n\) with \(P_4'_{1} = P_4''\ \text{CONSTRUCT} R_4'.\)

The evaluation of the basic pattern \(P_4''\) over \(G\) gives

\([P_4''] = \{\text{auth1 publishes} \ ?m1, \text{auth2 likes} \ ?m1, \text{auth1 publishes} \ ?m2, \text{auth2 likes} \ ?m2\}\) and:

\[
\begin{array}{|c|c|c|}
\hline
?a & ?m & ?a2 & ?m2 \\
\hline
\text{auth1} & \text{mes1} & \text{auth2} & \text{mes3} \\
\text{auth2} & \text{mes3} & \text{auth1} & \text{mes1} \\
\hline
\end{array}
\]
Then for the subpattern $P_{4,1}'$ we get $[P_{4,1}'] = \{ \text{?a1 friend ?a2} \}$ and:

$$\text{Tab}([[P_{4,1}']]) = \begin{array}{cc}
\text{?a1} & \text{?a2} \\
\text{auth1} & \text{auth2} \\
\text{auth2} & \text{auth1}
\end{array}$$

so that $[P_4'] = \{ \text{?a1 friend ?a2 . ?n} \}$ and:

$$\text{Tab}([[P_4']]) = \begin{array}{ccc}
\text{?a1} & \text{?a2} & \text{?n} \\
\text{auth1} & \text{auth2} & 1 \\
\text{auth2} & \text{auth1} & 1
\end{array}$$

and finally $[P_4] = \{ \text{?a1 nb_of_friends ?n . ?n} \}$ and:

$$\text{Tab}([[P_4]]) = \begin{array}{cc}
\text{?a1} & \text{?n} \\
\text{auth1} & 1 \\
\text{auth2} & 1
\end{array}$$

so that

$$\text{Result}(Q_4, G) = \{ \text{auth1 nb_of_friends 1 . auth2 nb_of_friends 1} \}.$$

### A.5 Join: friendship relations

The subpattern $P_{4,1}'$ in Example A.4 builds the graph of friendship. Here is another pattern $P_5$ for the same purpose, which uses the symmetry of the friendship relation:

```sql
CONSTRUCT { ?a1 friend ?a2 }  
WHERE {  
    CONSTRUCT { ?a1 friend ?a2 }  
    WHERE { ?a1 publishes ?m . ?a2 likes ?m }
    JOIN { ?a2 friend ?a1 }
}
```

The value of the subpattern (let us call it $P_{5,1}$) is the set of matches from $[P_{5,1}] = \{ \text{?a1 friend ?a2} \}$ with assignment table:

$$\text{Tab}([[P_{5,1}]]) = \begin{array}{cc}
\text{?a1} & \text{?a2} \\
\text{auth1} & \text{auth2} \\
\text{auth2} & \text{auth1} \\
\text{auth3} & \text{auth1} \\
\text{auth3} & \text{auth2}
\end{array}$$

Then, the semantics of JOIN says that the basic pattern $P_{5,2} = \{ \text{?a2 friend ?a1} \}$ is not evaluated over $G$ but over the graph $G[[P_{5,1}]]$, which is $G$ extended with

$$\{ \text{auth1 friend auth2 . auth2 friend auth1 . auth3 friend auth1 . auth3 friend auth1 . auth3 friend auth2} \}.$$

Thus $[P_{5,2}] = \{ \text{?a2 friend ?a1} \}$ and:

$$\text{Tab}([[P_{5,2}]]) = \begin{array}{cc}
\text{?a2} & \text{?a1} \\
\text{auth1} & \text{auth2} \\
\text{auth2} & \text{auth1} \\
\text{auth3} & \text{auth1} \\
\text{auth3} & \text{auth2}
\end{array}$$
Here the join of both sets of matches is their intersection:

\[
\text{Tab}([P_{5,1}]) = \begin{bmatrix}
?a1 & ?a2 \\
\text{auth1} & \text{auth2} \\
\text{auth2} & \text{auth1}
\end{bmatrix}
\]

so that the result of query \(Q_5\) is the friendship relation:

\[
\text{Result}(Q_5, G) = \{\text{auth1 friend auth2 . auth2 friend auth1}\}.
\]

A.6 Union

The next query \(Q_6\) builds the graph of relation \(\text{foo}\), defined as follows: two authors are related by \(\text{foo}\) when at least one of them likes a publication by the other.

\[
\text{CONSTRUCT} \{ ?a1 \text{ foo } ?a2 \}
\]

\[
\text{WHERE} \{ ?a1 \text{ publishes } ?m1 . ?a2 \text{ likes } ?m1 \}
\]

\[
\text{UNION}
\]

\[
\text{CONSTRUCT} \{ ?a1 \text{ foo } ?a2 \}
\]

\[
\text{WHERE} \{ ?a2 \text{ foo } ?a1 \}
\]

It is easy to check that the result of query \(Q_6\) is:

\[
\{\text{auth1 foo auth2 . auth2 foo auth1 . auth1 foo auth3 . auth3 foo auth1 . auth2 foo auth3 . auth3 foo auth2}\}
\]

A.7 Fresh variables

Here the goal is to build, for each author \(?a\) and each message \(?m\) published by \(?a\) and stamped at date \(?d\), a tree with a fresh variable as root and with two branches, one named author towards \(?a\) and the other one named date towards \(?d\). The pattern \(P_7\) is:

\[
\text{CONSTRUCT} \{ ?r \text{ author } ?a . ?r \text{ date } ?d \}
\]

\[
\text{WHERE} \{ ?a \text{ publishes } ?m . ?m \text{ stampedAt } ?d \}
\]

Note that the variable \(?r\) in \(R_7\) does not occur in \(P_7\).

The graph \([P_7]\) is \{\(?a\) publishes \(?m\). \(?m\) stampedAt \(?d\)\} and:

\[
\text{Tab}([P_7]) = \begin{bmatrix}
?a & ?m & ?d \\
\text{auth1} & \text{mes1} & \text{date1} \\
\text{auth1} & \text{mes2} & \text{date2} \\
\text{auth2} & \text{mes3} & \text{date1} \\
\text{auth3} & \text{mes4} & \text{date4} \\
\text{auth3} & \text{mes5} & \text{date4}
\end{bmatrix}
\]

It follows that \([P_7] = \{?r \text{ author } ?a . ?r \text{ date } ?d\}\) and:

\[
\text{Tab}([P_7]) = \begin{bmatrix}
?r & ?a & ?d \\
?r1 & \text{auth1} & \text{date1} \\
?r2 & \text{auth1} & \text{date2} \\
?r3 & \text{auth2} & \text{date1} \\
?r4 & \text{auth3} & \text{date4} \\
?r5 & \text{auth3} & \text{date4}
\end{bmatrix}
\]
Note that the variable $?r$ in $R_7$, which is not a variable of $P'_7$, gives rise to one fresh variable for each match of $[P'_7]$ in $G$. In fact, the query $Q_7$ “mimicks” the following SELECT query $Q'_7$:

```
SELECT ?a ?d
WHERE { ?a publishes ?m . ?m stampedAt ?d }
```

As explained in [4], the various copies of the variable $?r$ in the result of $Q_7$ act as identifiers for the rows in the table result of $Q'_7$ over $G$ (as for instance in SPARQL), which is obtained by dropping the column $?r$ from $Tab([[[P_7]]]_G)$. Note that the table $Tab([[[P_7]]]_G)$ has all its rows distinct by definition, whereas this becomes false when the column $?r$ is dropped.