Extending the SPARQL Algebra for the optimization of Property Paths

Louis Jachiet, Pierre Genevès, Nils Gesbert, Nabil Layaïda

Extending the sparql Algebra for the optimization of Property Paths

Louis Jachiet
Univ. Grenoble Alpes, CNRS, Grenoble INP*, LIG,
38000 Grenoble, France
louis.jachiet@inria.fr

Pierre Genevès
Univ. Grenoble Alpes, CNRS, Grenoble INP*, LIG,
38000 Grenoble, France
pierre.geneves@cnrs.fr

Nabil Layaïda
Univ. Grenoble Alpes, CNRS, Grenoble INP*, LIG,
38000 Grenoble, France
nabil.layaida@inria.fr

Nils Gesbert
Univ. Grenoble Alpes, CNRS, Grenoble INP*, LIG,
38000 Grenoble, France
nils.gesbert@grenoble-inp.fr

ABSTRACT

SPARQL is the W3C standard language for querying RDF graphs. Since its 1.1 version, SPARQL allows queries with Property Paths. They correspond to a form of regular path queries over graphs that raises new challenges for SPARQL evaluators and RDF stores. Property Paths can be recursive and the optimization of recursive queries remains a challenge both in the relational world and in the semantic web world.

In this paper, we present μ-algebra, a variation of the SPARQL Algebra that allows for the optimization of SPARQL especially with Property Paths. Similarly to the SPARQL Algebra, SPARQL 1.1 queries can be translated into our algebra but the obtained μ-algebra terms can be rewritten into multiple equivalent terms. Each of these terms is a possible execution plan of the initial SPARQL query.

We then show that our method generates terms corresponding to execution plans that are not considered by existing methods. We experimentally demonstrate the advantages of our approach. We have implemented a prototype that translates, optimizes and evaluates SPARQL queries using our algebra. Experiments show that our prototype outperforms other existing methods on recursive queries.

ACM Reference Format:
Louis Jachiet, Pierre Genevès, Nabil Layaïda, and Nils Gesbert. 2018. Extending the SPARQL Algebra for the optimization of Property Paths. In *Proceedings of WWW*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

RDF is the W3C standard language for Semantic Web data and SPARQL is its companion language intended for the querying of RDF graphs. Since its 1.1 version, SPARQL introduced Property Paths. Property Paths are a feature to express Regular Paths Queries inside SPARQL queries.

While the SPARQL query optimization is already well-studied, the specific subject of the optimization of queries containing Property Paths raises many challenges that need to be overcome. As it has been noted by others [4] and as we will demonstrate in our theoretical comparison and practical experiments actual query optimizers cannot handle yet simple queries even for very simple queries and reasonably small graphs.

Finally, the ever growing amounts of available RDF data dictates the use of distributed methods and thus of query plans capable of scaling.

Contributions. In this context, we propose a new algebra inspired by works on the relational algebra, SQL and NoSQL languages (especially SPARQL). Our algebra subsumes the SPARQL Algebra (under the set semantics) with a more general recursion. We introduce a translation from SPARQL with Property Paths to this algebra. We introduce a type system and new rewriting rules that allow optimization of terms involving fixpoint operators. We illustrate the benefits of our approach on recursive query optimization and report on experimental results illustrating the practical feasibility. While a generic approach often comes at the cost of performance, we experimentally show that this approach leads to more efficient evaluation of queries with Property Paths. We also show that our approach produces Query Execution Plans (QEP) that are not considered by other existing approaches.

2 SPARQL WITH PROPERTY PATHS

SPARQL[1] is a W3C standard to query RDF graphs. SPARQL and RDF recommendations come with an extensive set of features: blank nodes, literals, types, localization support, nesting of expressions, named graphs, etc. However from an evaluation point of view, most of these features can be hidden into user-defined functions of more general operators which greatly simplifies the optimization of SPARQL query evaluation. For instance checking the language of literal or checking whether a node is a blank can be done using special kind of filter and thus the optimizer only sees filters.

[1] https://www.w3.org/TR/SPARQL11-query/

WWW, 2018, Lyon

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1 rdf graphs

The RDF specification distinguishes unique resource identifiers (URI), blank nodes and literals. We note U the set of valid URI, blank node names and literals. An RDF graph is thus a set $T \subseteq U \times U \times U$ (notice that not all subsets of U^3 are valid RDF graphs since e.g. literals cannot as predicate or subjects).

Each triple $(s, p, o) \in T$ indicates that there is an edge from the node s to the node o labeled by p. In the RDF terminology s is the Subject, o is the Object and p the Predicate.

RDF data can also include a "context" information (or named graph). This name is a URI and therefore RDF data with named graphs can be represented as a quadruple of data $Q \subseteq U^4$ with $(s, p, o, g) \in Q$ being: s the subject, p the predicate, o the object and g the name of the graph.

2.2 Mappings

Solutions of a SPARQL query are sets (or bags) of "mappings". A *mapping* is a function from strings to strings with a finite domain. Formally, a mapping corresponds to a set of the form $\{k_i \to v_i \mid i \in I\}$ with I finite and the $(k_i)_{i\in I}$ all distinct. The set $\{k_i \mid i \in I\}$ is called the *domain* of the mapping, written $dom(\{k_i \to v_i \mid i \in I\})$.

We say that two mappings a and b are *compatible* (written $a \sim b$) when $a(k) = b(k)$ for all $k \in dom(a) \cap dom(b)$. Given two compatible mappings $a = \{k_1 \to v_1, \ldots, k_n \to v_n\}$ and $b = \{k'_1 \to v'_1, \ldots, k'_n \to v'_l\}$, we can combine them into $a + b = a \cup b$ (it defines a mapping since $a \sim b$).

In terms of the usual tabular presentation of query results, the domains of the solutions correspond to *column* names (and are strings), while each mapping is a *row*.

2.3 Triple patterns

The building blocks of SPARQL queries are the Triple Patterns (TP). In addition to the set U of names used in an RDF graph we suppose a set V of Variables names (disjoint from U). A TP if a triple $(s, p, o) \in (U \cup V)^3$. The set of variables of a TP is $var(s, p, o) = \{s, p, o\} \cap V$.

A TP (s, p, o) *matches* a triple $(s', p', o') \in T$ if we can replace the variables of (s, p, o) to obtain (s', p', o'). Formally, (s, p, o) *matches* (s', p', o') when there is a mapping m, with $dom(m) = var(s, p, o)$ and such that $(s', p', o') = (\tilde{s}, \tilde{p}, \tilde{o})$ where $\tilde{x} = x$ when $x \in U$ and $\tilde{x} = m(x)$ when $x \in V$; the mapping m is the *solution* of this matching.

The set of solutions for a TP against an RDF graph is the set of mappings that are solutions of matchings between the TP and the triples composing the graph.

2.4 Property paths

SPARQL 1.1 introduced Property Paths (PP). The idea behind PP was to extend TP. Instead of matching labeled arcs (as done with TP in SPARQL 1.0) we can also match paths in the graph where the sequence of labels of the edges of the path matches a given regular expression.

The syntax of PP is presented in figure 3. Note that, in the W3C standard, variables can appear in TP but not in PP; we include them to treat TP and PP alike.

U	::=	$URI \mid blank\ node \mid literal$	
V	::=	$?varname$	
UV	::=	Constant(U) — Variable(V)	
RPE	::=		*Regular Path Expression*
	|	UV	Constant & Variables
	|	(RPE/RPE)	Concatenation
	|	$(RPE \mid RPE)$	Disjunction
	|	(RPE^{-1})	Reverse
	|	$(RPE?)$	Potential
	|	$(RPE*)$	Transitive reflexive closure

Figure 1: Syntax of property paths.

2.5 Filters

SPARQL is equipped with filters. We do not present here the rich language of filter conditions included in SPARQL since we will manipulate them as black boxes. We only suppose that they are such that we can compute a set of column names that might be used, we write $FV(f)$ to compute the set of column names that the filter f might consider (i.e. $FV(f)$ is such that a mapping m passes the filter f if and only if the mapping m' which is m with a domain reduced to $FV(f)$ passes the filter: for any mapping m and filter f, we have $f(m) = f(m_{FV(f)})$ where $m_{FV(f)} = \{(k \to v) \in m \mid k \in FV(f)\}$).

As important kind of filter, we have: the test of column presence (written $bnd(c)$) which evaluates to true on mappings having c in their domains; its opposite (written $\neg bnd(c)$) that evaluates to true on mappings not having c in their domain.

2.6 sparql

We now present the SPARQL-Algebra. The SPARQL Algebra is an abstract tree representation of SPARQL queries and is now part of the W3Cx standard[2]. Several SPARQL query evaluators (such as Jena [3]) use a variant of this algebra as an internal representation of queries and several articles already deal with the semantics, complexity [14], the optimization and the rewriting of SPARQL Algebra terms [17].

SPARQL and SPARQL Algebra contain "solution modifiers" to: sort the data, limit the number of solutions, etc. One important such solution modifier is the DISTINCT keyword.

The SPARQL Algebra usually comes in two flavors: the "set" semantics and the "bag" semantics. In the "bag" semantics, solutions are counted with multiplicity. In the SPARQL standard, the default semantic is the "bag" semantic except for paths queries (to deal with cyclic paths) and this default can be overridden with the keyword DISTINCT, either at the top level or on subqueries.

[2] https://www.w3.org/2001/sw/DataAccess/rq23/rq24-algebra.html
[3] https://jena.apache.org/

$$
\begin{array}{llr}
GP & ::= & \textit{Graph Pattern} \\
 & | \ PP(UV, RPE, UV) & \\
 & | \ Graph(name, GP) & \\
 & | \ Filter(cond, GP) & \\
 & | \ Join(GP, GP) & \\
 & | \ Diff(cond, GP, GP) & \\
 & | \ LeftJoin(GP, GP) & \\
 & | \ Union(GP, GP) & \\
 & | \ Minus(GP, GP) & \\
 & | \ FilterNotExists(GP, GP) &
\end{array}
$$

Figure 2: sparql Algebra without query modifiers.

For the sake of simplicity, we present the SPARQL Algebra only for the "set" semantics of the SPARQL Algebra and with no solution modifiers but the "bag" semantic can easily be obtained using the aggregation operator available in our extended SPARQL Algebra.

The abstract syntax of the SPARQL Algebra we consider is presented in figure 2. GP stands for Graph Pattern, $cond$ represents here a condition that can be expressed in a SPARQL filter and $name$ in $Graph(name, GP)$ stands for a valid graph name.

In the official recommendation from the W3C, there is no operator PP but there is an operator BGP (for Basic Graph Pattern). BGP are composed of TP and PP. The semantic of a BGP over the n triple patterns (or PP) $tp_1 \ldots, tp_n$ is simply the natural join of the solutions of each triple pattern. Therefore BGP can be represented using the $Join$ operator and our PP. More precisely, $BGP(tp_1. \ldots .tp_n)$ can be seen as syntactic sugar for $Join(PP(tp_1), Join(\ldots, PP(tp_n)) \ldots)$.

3 THE μ-ALGEBRA

3.1 Origins of μ-algebra

μ-algebra is a language heavily inspired by the SPARQL Algebra and the relational algebra. Being at the crossroads between the SPARQL and the relational algebras, μ-algebra borrows features from both. μ-algebra can be seen as a variant of the relational algebra using several traits specific to the SPARQL Algebra: the filters, the user-defined functions, the handling of null values.

One notable line of work that inspired μ-algebra is the work on equipping the relational algebra with transitive closures (see e.g. [1]) and the optimization of the resulting language ([13]).

The μ-algebra algebra also has a variable binding operator ($let\ X = \varphi\ in\ \psi$). This operator does not add expressiveness (it can be replaced by the formula ψ' which is ψ where free occurrences of X are replaced with φ) but it allows us to sometimes limit the combinatory explosion that might arise by expanding let-bindings. Finally, we have a fixpoint operator ($\mu X = \varphi$) that allows us to capture the newest SPARQL features (e.g. Property Paths) and make optimizations that are tailored for SPARQL use-cases.

$$
\begin{array}{llr}
\varphi & ::= & \text{formula} \\
 & | \ \varphi_1 \cup \varphi_2 & \text{union} \\
 & | \ \varphi_1 \backslash\!\backslash\, \varphi_2 & \text{normal minus} \\
 & | \ \varphi_1 - \varphi_2 & \text{set minus} \\
 & | \ \varphi_1 \setminus \varphi_2 & \text{strict minus} \\
 & | \ \varphi_1 ⟕ \varphi_2 & \text{left-join} \\
 & | \ \varphi_1 \bowtie \varphi_2 & \text{join} \\
 & | \ \rho_a^b(\varphi) & \text{exchange column (or rename)} \\
 & | \ \pi_a(\varphi) & \text{column dropping (projection)} \\
 & | \ \beta_a^b(\varphi) & \text{column multiplying} \\
 & | \ \theta(\varphi, f : \mathcal{C} \to \mathcal{D}) & \text{apply a function to mappings} \\
 & | \ \Theta(\varphi, f : \mathcal{C} \to \mathcal{D}) & \text{reduce} \\
 & | \ \sigma_f(\varphi) & \text{row filtering} \\
 & | \ \mu(X = \varphi) & \text{fixpoint} \\
 & | \ \text{let } (X = \varphi) \text{ in } \psi & \text{let-binder} \\
 & | \ X & \text{variable} \\
 & | \ \emptyset & \text{no mapping} \\
 & | \ |c_1 \to v_1, \ldots, c_n \to v_n| & \text{constant}
\end{array}
$$

Figure 3: Grammar of μ − sparql

3.2 The language

The abstract syntax of the language is presented in Figure 3 and its semantics is given in figure 4. We give here the intuition behind the μ-algebra. Basic μ-algebra terms are variables and terms can be combined or modified via the following operators:

- $\varphi_1 \cup \varphi_2$ Union of the sets of mappings from φ_1 and φ_2;
- $\varphi_1 \bowtie \varphi_2$ Join operator: the combination of pairs of mappings from φ_1 and φ_2 that are compatible;
- $\varphi_1 ⟕ \varphi_2$ Left-join: the combination of pairs of compatible mappings from φ_1 and φ_2 plus the mappings of φ_1 not compatible;
- $\varphi_1 \setminus \varphi_2$ Strict Minus: the mappings from φ_1 for which there are no compatible mapping in φ_2; it corresponds to the $FilterNotExists$ in the SPARQL Algebra;
- $\varphi_1 \backslash\!\backslash\, \varphi_2$ Normal Minus: the mappings from φ_1 for which all compatible mappings in φ_2 have a disjoint domain; it corresponds to the $Minus$ in the SPARQL algebra;
- $\varphi_1 - \varphi_2$ Set Minus: the mappings from φ_1 that do not appear as mappings of φ_2; it is used for translation of the conditional $Diff$ of the SPARQL Algebra, see section 4.4.
- $\rho_a^b(\varphi)$ Exchange a and b in the domain of the mappings of φ; when a (resp. b) does not exist then this operator just renames b into a (resp. a into b);
- $\pi_a(\varphi)$ Remove the column a from the mappings of φ;
- $\beta_a^b(\varphi)$ The effect of this operator is that in all mappings of φ, b is mapped to the same value as a (or no value if a is not in the mapping).
- $\theta(\varphi, f : \mathcal{C} \to \mathcal{D})$ Map the mappings m of φ through the user-defined function f. The notation $f : (\mathcal{C} \to \mathcal{D})$ indicates that when f is given a mapping over a domain $M \supset \mathcal{C}$ it returns a mapping over the domain $M \cup \mathcal{D}$. When $\mathcal{C} \not\subseteq M$, f leaves the mapping unchanged (we suppose $M \cap \mathcal{D} = \emptyset$);

$$
\begin{array}{rcl}
[\![\varphi_1 \bowtie \varphi_2]\!]_V & = & \{a+b \mid a \in [\![\varphi_1]\!]_\wedge \ b \in [\![\varphi_2]\!]_\wedge \ a \sim b\} \\
[\![\varphi_1 \backslash\backslash \varphi_2]\!]_V & = & \{m \in [\![\varphi_1]\!]_V \mid \nexists n \in [\![\varphi_2]\!]_V \ n \sim m \wedge dom(n) \cap dom(m) \neq \emptyset\} \\
[\![\varphi_1 \setminus \varphi_2]\!]_V & = & \{m \in [\![\varphi_1]\!]_V \mid \nexists n \in [\![\varphi_2]\!]_V \ n \sim m\} \\
[\![\varphi_1 - \varphi_2]\!]_V & = & \{m \in [\![\varphi_1]\!] \mid m \notin [\![\varphi_2]\!]_\}\\
[\![\varphi_1 \bowtie\!\!\!\!- \ \varphi_2]\!]_V & = & [\![\varphi_1 \bowtie \varphi_2]\!]_V \cup [\![\varphi_1 \setminus \varphi_2]\!]_V \\
[\![\pi_a(\varphi)]\!]_V & = & \{\{c \to v \mid (c \to v) \in m \wedge c \neq a\} \mid m \in [\![\varphi]\!]_V\} \\
[\![\beta_a^b(\varphi)]\!]_V & = & \{\{c \to v \mid (c \neq b \wedge c \to v \in m) \vee (c = b \wedge a \to v \in m)\} \mid m \in [\![\varphi]\!]_V\} \\
[\![\rho_a^b(\varphi)]\!]_V & = & \{\{c \to v \mid (c = a \wedge b \to v \in m) \vee (c = b \wedge a \to v \in m) \vee (c \neq a \wedge c \neq b \wedge c \to v \in m)\} \mid m \in [\![\varphi]\!]_V\} \\
[\![\sigma_f(\varphi)]\!]_V & = & \{m \mid m \in [\![\varphi]\!]_V \ \wedge eval(f[m]) = \top\} \text{ (see SPARQL semantics for } eval(f[m])) \\
[\![c_1 \to v_1, \ldots, c_n \to v_n|]\!]_V & = & \{\{c_1 \to v_1, \ldots, c_n \to v_n\}\} \\
[\![\Theta(\varphi, f : \mathcal{C} \to \mathcal{D})]\!]_V & = & \{m + f(M) \mid \mathcal{C} \cap dom(m) = \emptyset \text{ and } M = \{m' \mid m + m' \in [\![\varphi]\!]_V \text{ and } dom(m') \subseteq \mathcal{C}\} \text{ and } M \neq \emptyset\}
\end{array}
$$

$$
\begin{array}{rclrcl}
[\![\emptyset]\!]_V & = & \emptyset & [\![\varphi_1 \cup \varphi_2]\!]_V & = & [\![\varphi_1]\!]_V \cup [\![\varphi_2]\!]_V \\
[\![X]\!]_V & = & V(X) & [\![\text{let } (X = \varphi) \text{ in } \psi]\!]_V & = & [\![\psi]\!]_{V[X \to [\![\varphi]\!]_V]} \\
[\![\theta(\varphi, f : \mathcal{C} \to \mathcal{D})]\!]_V & = & \{m + f(m)\} & [\![\mu(X = \varphi)]\!]_V & = & [\![X]\!]_{V[X \to U]}
\end{array}
$$

Where $U_0 = \emptyset$, $U_{i+1} = U_i \cup [\![\varphi]\!]_{V[X \to U_i]}$ and $U = lim_{n \to \infty} U_i$.

Figure 4: Semantics of μ-algebra

$\Theta(\varphi, f : \mathcal{C} \to \mathcal{D})$ Corresponds to a reduce operation using the user-defined function f and where the key is the complement of $\mathcal{C}$; i.e. two (or more) mappings m_1, m_2 are mapped to the same value when $dom(m_1) \setminus \mathcal{C} = dom(m_2) \setminus \mathcal{C}$ and $\forall c \in dom(m_1) \setminus \mathcal{C} : m_1(c) = m_2(c)$; $\Theta(\varphi, f : \mathcal{C} \to \mathcal{D})$ can be seen as the reduce or combine where $\theta(\varphi, f : \mathcal{C} \to \mathcal{D})$ is the map in a map-reduce setup; similarly to $\theta(\varphi, f : \mathcal{C} \to \mathcal{D})$, we have $f(M) \subseteq \mathcal{D}$;

let $X = \varphi$ *in* ψ Bind the variable X with a set of mappings of φ then returns the set of mappings of ψ ;

$\sigma_f(\varphi)$ Keep the mappings m such that $f(m)$ evaluates to true;

$\mu(X = \varphi)$ It corresponds to the fixpoint of the function $S \to S \cup [\![\psi]\!]_{V[X/S]}$ (where $[\![\psi]\!]_{V[X/S]}$ is the semantic of ψ where the variable X is replaced by the set of mappings S). We suppose in $\mu(X = \varphi)$ that φ is increasing in X (i.e. $A \subseteq B \Rightarrow [\![\varphi]\!]_{V[X \to A]} \subseteq [\![\varphi]\!]_{V[X \to B]}$); that can be forced as a syntatic criterion: no $\bowtie\!\!\!\!-$, $\backslash\backslash$, $-$, $\setminus$ or $\Theta(\psi, f : \mathcal{C} \to \mathcal{D})$ can appear in φ.

3.3 Filter, Aggregate, Map

Our μ-algebra language is equipped with three types of user-defined functions, the f appearing in each of these terms: $\sigma_f(\varphi)$, $\theta(\varphi, f : \mathcal{C} \to \mathcal{D})$, $\Theta(\varphi, f : \mathcal{C} \to \mathcal{D})$.

From the point of view of the μ-algebra, we consider these functions as user-defined but in practice they correspond to the rich variety of operations possible in SPARQL however as they come from a simple translation we easily express simple properties on those functions. For instance for f appearing in a $\sigma_f(\varphi)$ we can list the set of column $FV(f)$ such that the filter actually depends on. For a map $\theta(\varphi, f : \mathcal{C} \to \mathcal{D})$ (resp. an aggregate $\Theta(\varphi, f : \mathcal{C} \to \mathcal{D})$), we suppose that f only reads the $\mathcal{C}$ part of the mappings and modify the $\mathcal{D}$ part.

4 TRANSLATION OF SPARQL QUERIES INTO μ-ALGEBRA TERMS

4.1 Encoding of the graph structure

A RDF graph G is given by a set of triples, $G \subset U \times U \times U$. When $(s, p, o) \in G$ we say that the node s is linked to the node o via an edge labeled by p.

The default encoding of the graph is to use a unique variable Q that contains one mapping per quadruple, each of these mappings has for domain: 's' for subject, 'p' for predicate, 'o' for object and 'g' for graph.

We also have a term T_n for each graph named n and a variable T referring to the default graph. All of them bind three columns: 's', 'p', 'o' (for subject, predicate and object).

Finally we have a variable N for the set of all nodes of the graph. Each node is represented by a mapping whose domain is $\{s\}$ (even though N also contains nodes that only appear as objects).

Note that we can derive all of these variables from Q: for each graph name *name*, we use $let\ T_{name} = \pi_g(\sigma_{g=name}(Q))\ in \ldots$ and as SPARQL allows us to change the graph queried, during the course of the translation we maintain T to be the set of triples (s, p, o) corresponding to the current graph using let binders and finally, $N = \rho_s^o(\pi_p(\pi_s(T))) \cup \pi_p(\pi_o(T))$.

4.2 Regular Path Expressions

The hard part of translating a PP to a μ-algebra term is the translation of the regular path expression. The basic idea is to recursively translate the regular path expression r to a μ-algebra term such that each mapping solution of this term maps s (resp. o) to s' (resp. o') with s' linked to o' via a path accepted by r. However, since we included variables in regular path expressions, the translation might bind more variables than just s and o. The translation of a regular path expression r is $rpe(r)$ as presented in figure 5.

$$
\begin{array}{rcl}
rpe(Constant(u)) & = & \pi_p\left(\sigma_{p=''u''}(T)\right) \\
rpe(Variable(v)) & = & \rho_p^v\left(T\right) \\
rpe((r_1/r_2)) & = & \pi_m\left(\rho_o^m\left(rpe(r_1)\right) \bowtie \rho_p^m\left(rpe(r_2)\right)\right) \\
rpe((r_1|r_2)) & = & rpe(r_1) \cup rpe(r_2) \\
rpe((r^{-1})) & = & \rho_s^o\left(rpe(r)\right) \\
rpe((r?)) & = & rpe(r) \cup \beta_s^o\left(N\right) \\
rpe((r*)) & = & \mu(X = \beta_s^o\left(N\right) \cup \pi_m\left(\rho_o^m\left(X\right) \bowtie \rho_s^m\left(rpe(r)\right)\right))
\end{array}
$$

Figure 5: Translation of TP.

4.3 Property Paths

We note $gp(a)$ our translation of the SPARQL Algebra term a into μ-algebra. The translation of the SPARQL Algebra term $PP(s, r_p, o)$ (where r_p is a regular path expression) into $gp(PP(s, r_p, o))$ is done in two steps: first, we obtain a μ-algebra term $rpe(r_p)$ representing the set of mappings solution of r_p; then, there are four cases depending on whether s is a value or a variable and whether t is a value (w) or a variable ($?w$). With $p = pp(r_p)$, $gp(PP(s, r_p, o))$ is equal to:

	$o = Constant(w)$	$o = Variable(?w)$
$s = Constant(v)$	$\pi_s\left(\pi_o\left(\sigma_{s=v}(\sigma_{o=w})(p)\right)\right)$	$\pi_s\left(\sigma_{s=v}\rho_o^{?w}\left(p\right)\right)$
$s = Variable(?v)$	$\pi_o\left(\sigma_{o=w}(\rho_s^{?v}\left(p\right))\right)$	$\rho_o^{?w}\left(\rho_s^{?v}\left(p\right)\right)$

In this translation, we implicitly supposed that if s or o are variables then they are different from each other and they do not appear in r_p. In case $o = Variable(?v)$ and $?v$ appears somewhere else in the PP (in s or in r_p), we replace it with $?v'$: we translate the PP $(s, r_p, Variable(?v'))$ to a μ-algebra term φ and finally the translation is $\pi_{v'}\left(\sigma_{v=v'}(\varphi)\right)$. Similarly, if $s = Variable(?w)$, and $?w$ appears in r_p, we replace it with $?w'$.

4.4 Translation of sparql algebra terms

SPARQL-Algebra terms are defined recursively and are very similar by design to our μ-algebra terms. A SPARQL-Algebra term can therefore be easily translated into our μ-algebra: we have seen the translation for PP; Join can be translated to our join (i.e. $gp(Join(a, b)) = gp(a) \bowtie gp(b)$); filters translate to our filter operator ($gp(Filter(a, f)) = \sigma_f(gp(a))$); unions translate to our union ($gp(Union(a, b)) = gp(a) \cup gp(b)$); graph selections translate to let bindings $gp(Graph(name, a)) = let\ T = T_{name}\ in\ gp(a)$; $FilterNotExists$ translate to a strict-minus ($\backslash$), $Minus$ to a minus ($\backslash\backslash$). Only the conditional difference is non-trivial to translate .

Conditional difference ($Diff(\Omega_1, \Omega_2, cond)$ in the W3C literature) is defined as the set of mappings m_1 from Ω_1 such that all compatible mappings m_2 from Ω_2 are such that $cond(m_1 + m_2)$ is false. Therefore $Diff(\Omega_1, \Omega_2, cond) = \Omega_1 - \Omega'$ where Ω' captures the mappings n from Ω_1 such that there is a mapping m from Ω_2 with $cond(n + m)$. This Ω' can be computed with terms of our algebra but we need to compute first which columns can appear in solutions of Ω_1 and Ω_2. See appendix for a complete translation.

Finally, there as terms not present in the SPARQL Algebra but present in the SPARQL standard, notably $Bind$ and the aggregate functions. These functions translate easily into $c_1 \to v_1, \ldots, c_n \to v_n$, $\theta(\varphi, f : \mathcal{C} \to \mathcal{D})$ and $\Theta(\varphi, f : \mathcal{C} \to \mathcal{D})$.

4.5 Example of translation

In the case of :

```
@prefix ex: <http://example/>
SELECT * WHERE {
  ?x ex:knows* ?y .
  ?x ex:named ex:bob . }
```

In our formulas, we remove the prefix part for simplicity. : First, this query is translated to the SPARQL Algebra: $Join(PP(?y\ named\ bob), PP(?x\ knows*\ ?y))$ then each PP is translated: $rpe(?y\ named\ bob) = \pi_o\left(\sigma_{o=bob}(\pi_p\left(\sigma_{p=named}(T)\right))\right)$ and $rpe(?x\ knows*\ ?y) = \rho_s^a\left(\rho_o^b\left(\mu(X = \beta_s^o\left(N\right) \cup \pi_m\left(\rho_o^m\left(X\right) \bowtie \rho_s^m\left(\pi_p\left(\sigma_{p=knows}(T)\right)\right)\right)\right)\right)\right)$ Finally, the total term is: $rpe(?x\ knows*\ ?y) \bowtie rpe(?y\ named\ bob)$.

5 ANALYSIS OF μ-ALGEBRA TERMS

We now present our type system for μ-algebra. This type system will be used in the next section in the definition of our rewriting rules. For instance, in a μ-algebra term like $\sigma_f(A \bowtie B)$ we can sometimes push the filter into A (i.e. it is equivalent to the term $\sigma_f(A) \bowtie B$). Our type system will allow us to detect situations where such rewritings are sure to be correct. This analysis is similar to other analyses in the context of the SPARQL Algebra, see e.g. the definition of $cVars$ and $pVars$ [18] that resembles our C and P, but we since our algebra has variables and fixpoints the typing system has been largely sophisticated.

5.1 Types

Our types are pairs (c, p) such that $c \subseteq V$ and $p \subseteq V$. Intuitively, a type (c, p) describes a set c of columns that are **c**ertain to be present in all mappings and a set p of columns that appear **p**ossibly. Formally, the pair (c, p) is a valid type for a set of mappings M when for each mapping $m \in M$ we have $c \subseteq dom(m) \subseteq p$.

Note that when (c, p) is the valid type for a set of mappings M then all (c', p') such that $c' \subseteq c$ and $p \subseteq p'$ are also valid types for M.

5.2 Types of μ-algebra terms

Given an environment V and a μ-algebra term φ, the type of $[\![\varphi]\!]_V$ depends, obviously, on φ but also on V. Our typing will therefore *abstract environment*. An abstract environment Γ is a mapping from variables to types and Γ abstracts V, written $\Gamma \vdash V$, when for all X, $V(X)$ is of type $\Gamma(X)$.

Our typing mechanism associates to each subterm φ a pair $\Gamma_\varphi, T_\varphi$ where φ is only evaluated in an environment V abstracted by Γ_φ and T_φ is a valid type for $[\![\varphi]\!]_V$ when $\Gamma_\varphi \vdash V$.

For instance, the term $T \cup \rho_s^a\left(\rho_p^b(T)\right)$ is typed $\Gamma, (\{o\}, \{a, b, s, p, o\})$ where Γ is the default abstract environment for μ-algebra terms (i.e. $\Gamma(T) = (\{s, p, o\}, \{s, p, o\})$).

5.3 Precision of types

The more precise the analysis, the more rewriting we can do and therefore the more optimization we get. Deciding the most precise type (i.e. decide whether a given column is sure to be present or will never be present) is a hard problem. For instance, it implies to decide whether or not a μ-algebra expression returns an empty set, which is, at least, as hard as deciding whether a SPARQL expression has a solution (actually even more since we allow arbitrary functions in filters).

That is why we use only a syntactic (therefore imprecise) criterion to compute types. These approximations are conservative, for all environments V with $\Gamma_\varphi \vdash V$ then a mapping m of $[\![\varphi]\!]_V$ will be such that $c \subseteq dom(m) \subseteq p$ where $(c, p) = T_\varphi$.

5.4 Analysis of fixpoints

From now on, we suppose that the type of formula is computed (and eventually recomputed after each rewriting). Notice that two syntactic equivalent terms can be evaluated into two different contexts and are therefore not considered equivalent anymore in the rest of this paper.

Similarly to Datalog, in μ-algebra we can define a notion of terms *constant* in a variable X, *linear* in a variable X or *recursive* in a variable X.

Intuitively, a term φ is *constant* in X when X is not a free variable of φ, φ is *linear* in X when all the free occurrences of X in φ are positive (i.e. not below the right side of a minus or a left join, not below an aggregation) and there are no join with X appearing as subformula of both side. Finally φ is *recursive* when φ has no constant part (i.e. when $\forall S : [\![\varphi]\!]_{V[X/S]} = \emptyset$).

5.4.1 Filters columns. Given a fixpoint $\mu(X = \varphi)$, we can compute a set $F(\varphi)$ of columns for which a filter operating of those columns commutes with the μ; or, formally, such that for all $c \in F(\varphi)$,all v and all V with $\Gamma_\varphi \vdash V$, $[\![\sigma_{c=v}(\mu(X = \varphi))]\!]_V = [\![\mu(X = \sigma_{c=v}(\varphi))]\!]_V$.

5.4.2 Addable columns. We can also compute a set $A(\varphi)$ of columns such that can be added to a fixpoint, i.e. $A(\varphi) \cap p_\varphi = 0$ with $c_\varphi, p_\varphi = T_\varphi$ and for all V such that $\Gamma_\varphi \vdash V$ then for all $c \in A(\psi), v$ then $[\![\mu(X = \varphi) \bowtie |c \to v|]\!]_V = [\![\mu(X = \varphi \bowtie |c \to v|)]\!]_V$.

5.4.3 Decomposition. The terms produced by our SPARQL translation are all linear in all recursive variables (recursive variable are the variables introduced by fixpoints). Given a fixpoint $\mu(X = \varphi)$ we can often decompose φ into $\varphi = init \cup rec$ where $init$ is constant and rec recursive in X.

6 REWRITING μ-ALGEBRA TERMS

The goal of this section is to present our mechanism that allows obtaining multiple μ-algebra terms that compute the same set of mappings as a given μ-algebra term. The objective afterwards is to choose between all the equivalent forms the one that is the most efficient. The determination of the most efficient is the subject of the next section.

The general idea used to produce those equivalent forms is similar to traditional optimization techniques used in e.g. SQL query optimizers: we have a set of *normalizing rules* and a set of *rewriting rules*. Each of these rules detect patterns in a μ-algebra that could be replaced by some other pattern. Each use of a *rewriting rule* produce a new term equivalent to the original term and we normalize this term using the normalizing rules (a term is normalized when no normalizing rule can be applied).

With a very general set of rewriting rules, the termination of the above algorithm is not certain. For instance, with only the valid rule $A \to A \cup \emptyset$, our algorithm will never terminate. So we have to make sure that such behaviour does not arise.

Furthermore, even with a terminating set of rules, the running time of our algorithm might prevent effective use on real queries. That is why we use *normalizing* rules, to reduce the number of μ-algebra terms considered.

There are a lot of rules described in the literature for rewriting relational queries or SPARQL Algebra terms. We will not present here the extensive set of such rewritings but refer the reader to previous works (see e.g. figure 2 of [17] or appendix) for a comprehensive set of such rules with their proof and we will sum up the general effect of these rules.

6.1 Normalization

Our normalization rules have the following effects:

- formulas containing $\emptyset$ as subformula can be simplified to $\emptyset$ or to a term that does not contain $\emptyset$ as subformula.
- the renamings (i.e. $\rho_a^b(\varphi)$) can be pushed downward so that they only appear in front of variables
- the column selections (i.e. $\pi_a(\varphi)$) can be pushed upward so that they only appear at the top.
- let binders (i.e. let $(X = \varphi)$ in ψ) can be either removed of pushed upward so they only appear at the top, below renamings, or below fixpoints.
- terms using only constants are rewritten so that are of the form $a_1 \cup (a_2 \cup (\dots a_n) \dots)$ where each a_i is a constant term (i.e. of the form $|c_1 \to v_1, \dots, c_n \to v_n|$).
- we suppose that there exists an order $<$ on formulas (e.g. the lexicographical order on their string representation) and we impose (using commutativity) that formulas of the shape $\varphi_1 \bowtie \varphi_2$ or $\varphi_1 \cup \varphi_2$ are such that $\varphi_1 < \varphi_2$.
- finally the last step of normalization is the removal of terms appearing several times. For instance, if we have the term $A \cup A$ then we can normalize it to A; we also have A *minus* $A \to \emptyset$ (for any kind of *minus*), $A \bowtie A = A$ when $|p_A \setminus c_A| \leq 1$ with $((c_A, p_A) = T_A)$, etc.

6.2 Generating new μ-algebra terms

To general new μ-algebra term the idea is to exhibit conditions for associativity and distributivity between operators. This

includes traditional rules such as pushing filters into joins but also new rules concerning the new feature of our algebra: fixpoints. We will now present the rules for fixpoints and invite the reader to read the appendix for a complete set of rules.

6.2.1 Combining fixpoints and joins or filters. In our translation of SPARQL, given the term $PP(John\ knows*\ ?a)$ our algorithm will translate $knows*$ into a fixpoint computation and then filter to keep elements of the relation that have a $John$. Clearly, computing the whole binary relation $knows*$ before filtering is wasteful compared with computing directly the unary relation $knows*\ John$.

Notice that $?a$ might be more constrained than $John$ through other TPs and in this cases our computation should start from $?a$; starting from the fixed part of a PP is not always the best plan. Therefore, in order to generate the various plans, we need rules to move joins and filters inside fixpoints such as $\sigma_f(\mu(X = A)) \rightarrow \mu(X = \sigma_f(A)))$.

Clearly, such a rule is not always true. If we consider the translation of $PP(John\ knows*\ ?a)$ then given the μ-algebra term $A = \beta^o_s(N) \cup \pi_m(\rho^m_s(X) \bowtie \rho^m_o(\sigma_{p=knows}(T)))$, we have $\sigma_{s=John}(\mu(X = A)) \rightarrow \mu(X = \sigma_{s=John}(A))$ but we don't have $\sigma_{o=John}(\mu(X = A)) \rightarrow$
$\mu(X = \sigma_{o=John}(A))$. This is because our fixpoint computation uses the column o to produce new elements.

Our rules are (with $(c_\psi, p_\psi) = T_\psi$ and $FV(f)$ the set of columns used by f) :

$$\begin{array}{rcll} \sigma_f(\mu(X = \varphi)) & \rightarrow & \mu(X = \sigma_f(\varphi)) & \text{when } FV(f) \subseteq F(\varphi) \\ \psi \bowtie \mu(X = \varphi) & \rightarrow & \mu(X = \psi \bowtie \varphi) & \text{when } c_\psi \setminus c_\varphi = p_\psi \setminus c_\varphi \\ & & & \text{and } c_\varphi \cap p_\psi \subseteq F(\varphi) \end{array}$$

Furthermore, fixpoints are often composed of an initial part and a recursive part. If the fixpoint is $\mu(X = \varphi)$, with φ decomposed into $init \cup rec$ where $init$ is constant in X and rec is recursive in X then we can also have (same conditions):

$$\begin{array}{rcl} \sigma_f(\mu(X = init \cup rec)) & \rightarrow & \mu(X = \sigma_f(init) \cup rec) \\ \psi \bowtie \mu(X = init \cup rec) & \rightarrow & \mu(X = \psi \bowtie init \cup rec) \end{array}$$

6.2.2 Reversing fixpoints. As we have seen, pushing filters, selections and joins into a fixpoint $\mu(X = \varphi)$ depends on φ recursively uses φ. The translation of the regular path expression $knows*$ is $rep(knows*) = \mu(X = \beta^o_s(N) \cup \pi_m(\rho^m_o(X) \bowtie \rho^m_s(\sigma_{p=knows}(T))))$. In this fixpoint, we cannot push the filter $\sigma_{o=John}$ since o is used by X. This fixpoint starts with a set of couples $\beta^o_s(N)$ binding s and o, and at each iteration of the fixpoint, it will build new (s, o) by appending a mapping (m, o) validating $knows$ to a mapping (s, m) already built. However, to compute the same set of mappings, the fixpoint could have started from the same set and recursively appended a mapping (s, m) validating $knows$ to an already-built mapping (m, o) without using o.

Given a fixpoint of the form (with the a_i and the b_j all distinct):

$$\mu\Big(X = \psi \cup \pi_{b_1}\left(\dots \pi_{b_k}\left(\rho^{b_1}_{a_1}\left(\dots \rho^{b_k}_{a_k}(X)\right) \bowtie \varphi\right)\right)\Big)$$

with $c_\psi = p_\psi = \{a_1, \dots, a_k, d_1 \dots d_n\}$ and $P_\varphi = \{a_1, \dots, a_k, b_1, \dots, b_k\}$. The fixpoint starts from the solutions of ψ then for each mapping m ($dom(m) = p_\psi$) it appends a mapping $m2$ solution of φ (such that $m(a_i) = m2(b_i)$) by keeping the (d_i) from m and the (a_j) from $m2$. Its reverse starts from the set of mappings $\{a_1 \rightarrow v_1, \dots, a_k \rightarrow v_k, b_1 \rightarrow v_1, \dots, b_k \rightarrow v_k\}$ (for all possible $v_1, \dots, v_k$) then at each step of the iteration replace $b_1, \dots, b_k$ by new values. Finally once the fixpoint is reached, we remove the columns $b_1, \dots, b_k$ and join with ψ. This gives us the μ-algebra term:

$\pi_{b_1}\left(\dots \pi_{b_k}\left(\mu(X = A \cup B) \bowtie \rho^{b_1}_{a_1}\left(\dots \rho^{b_k}_{a_k}(\psi)\right)\right)\right)$ where $A =$
$\pi_{c_1}\left(\dots \pi_{c_k}\left(\rho^{c_1}_{a_1}(\dots \rho^{c_k}_{a_k}(X)) \bowtie \rho^{c_1}_{b_1}\left(\dots \rho^{c_k}_{b_k}(\varphi)\right)\right)\right)$
and $B = \beta^{b_1}_{a_1}\left(\dots \beta^{b_k}_{a_k}(\rho^{a_1}_s(N) \bowtie \dots \bowtie \rho^{a_k}_s(N)))\right)$

Such a rule also exists when the (a_i) are not all different from the (b_j) but we need to be careful when renaming the a_i into b_j.

6.2.3 Combine fixpoints. Given the Regular Path Expression $locatedIn*/sameAs*$ its translation will imply two fixpoints: one to compute $locatedIn*$ and one to compute $sameAs*$. Each of these fixpoints might have a very large number of solutions and joining them will be costly, it might have been more efficient to compute both in a single fixpoint: $\mu(X = \beta^o_s(N) \cup \pi_m(\rho^m_s(X) \bowtie loc \cup \rho^m_o(X) \bowtie sam))$ where $loc = \rho^m_o(\sigma_{p=locatedIn}(T))$ and $sam = \rho^m_s(\sigma_{p=sameAs}(T))$.

In general, if we have the join of two fixpoints operating on two separate domains then they can be joined: $\mu(X = init_1 \cup rec_1) \bowtie \mu(X = init_2 \cup rec_2)$ can be rewritten to $\mu(X = init_1 \bowtie init_2 \cup rec_1 \cup rec_2)$ when $p_{rec_1} \subseteq A(rec_2)$ and $p_{rec_2} \subseteq A(rec_1)$ (with $(p_{rec_i}, c_{rec_i}) = T_{rec_i}$).

6.2.4 Remove N. Computing N the whole set of nodes is often very costly in practice and, as we will see, can often be avoided. If we have a term $\sigma_{s=value}(N)$ then we will not simplifies this (but its execution is very fast, it returns a single value or the empty set).

When we have a term $N \bowtie \varphi$ and $s \in C(\varphi)$ then if we are sure that given a mapping m solution of φ then $m(s)$ is a node then we can rewrite this term to φ.

6.3 Example of rewriting

Given the SPARQL query:

```
SELECT * WHERE {
?a (knows)* ?b .
?a firstname John .
?b lastname Doe . }
```

Its translation is (1) $A_1 \bowtie B_1 \bowtie \rho^{?a}_s\left(\rho^{?b}_o(\mu(X = \beta^o_s(N) \cup C_1))\right)$ where:

$$A_1 = \pi_p\left(\rho^{?a}_s(\sigma_{o=\text{“}john\text{''}}(\sigma_{p=\text{“}firstname\text{''}}(T)))\right)$$

$$B_1 = \pi_s\left(\rho^{?b}_o(\sigma_{o=\text{“}Doe\text{''}}(\pi_p((\sigma_{p=\text{“}lastname\text{''}}(T)))))\right)$$

$$C_1 = \pi_m(\rho^m_o(X) \bowtie \rho^m_s(\pi_p(\sigma_{p=\text{”}knows\text{”}}(T))))$$

By pushing renamings so that they appear only in front of variables, we have (2) $A_2 \bowtie B_2 \bowtie \mu\big(X = \beta_s^{?b}\left(\rho_s^{?a}(N)\right) \cup C_2\big)$ where:

$$A_2 = \pi_p\left(\sigma_{o=\text{“}john''}(\sigma_{p=\text{“}firstname''}(\rho_s^{?a}(T)))\right)$$

$$B_2 = \pi_o\left(\sigma_{o=\text{“}Doe''}(\pi_p\left((\sigma_{p=\text{“}lastname''}(\rho_s^{?b}(T)))\right)\right)\right)$$

$$C_2 = \pi_m\left(\rho_{?b}^m(X) \bowtie \pi_p\left(\sigma_{p=\text{”}knows\text{”}}(\rho_o^{?b}(\rho_s^m(T)))\right)\right)$$

Then by combining the fixpoint with A_2 we have (3), $B_2 \bowtie \mu\big(X = \beta_{?a}^{?b}(A)_2 \cup C_2\big)$. We cannot combine the fixpoint with B from (2) since $?b$ of X is used in C (renamed to m then use on a join) but we can reverse the fixpoint which gives us (4), $A_2 \bowtie B_2 \bowtie \mu\big(X = \beta_s^{?b}\left(\rho_s^{?a}(N)\right) \cup C_3\big)$ where:

$$C_3 = \pi_m\left(\rho_{?a}^m(X) \bowtie \pi_p\left(\sigma_{p=\text{”}knows\text{”}}(\rho_s^{?a}(\rho_o^m(T)))\right)\right)$$

With this reversed fixpoint, we now can combine the fixpoint with B which gives us (5), $A_2 \bowtie \mu\big(X = \beta_{?b}^{?a}(B)_2 \cup C_3\big)$.

From a query execution point of view, the μ-algebra term (1) corresponds to evaluating each PP individually and then merge them. (3) starts by evaluating $?a$ $firstname$ $John$ then builds with a fixpoint the set of mappings verifying the two first PP and finally joins with the mappings solution of the third PP. (5) starts from the solution of the third PP, build the solution of the last PP with the fist and finally joins with the second PP.

7 OPTIMIZATION

The optimization of query evaluation depends on the plan space considered by the optimizer but it also depends on the capacity of the optimizer to select an efficient term among all the equivalent one, ideally the most efficient (even though finding the best is not always easy).

To compare the effectiveness, we devised a cost model for μ-algebra terms and a cardinality estimation of subterms.

7.1 Evaluation of μ-algebra terms

We implemented a prototype executor of μ-algebra terms. In this prototype, the execution time for terms is generally linear in the sum of the size of the results of its subterms plus a linear time for the output. However there are exceptions in some case of some operators on *unsafe* μ-algebra terms. A μ-algebra term t is *unsafe* when $P(t) \neq C(t)$.

For instance, joins are done by doing a hash-join: given the μ-algebra term $A \bowtie B$ we hash mappings on $c_A \cap c_B$. This work well when $p_A \cap p_B = c_A \cap c_B$ because mappings (m_A, m_B) solutions of $A \times B$ are compatible if and only if they have the same value on the domain $c_A \cap c_B$.

When $p_A \cap p_B \neq c_A \cap c_B$, the execution of a join is much more difficult. For each pair (m_A, m_B) of solution of $A \times B$, checking that they agree on the domain $c_A \cap c_B$ is easy but we also have to check that they agree on the domain $(p_A \cap p_B) \setminus (c_A \cap c_B)$ which can lead to a lot of checks without producing solutions. In the worst case, it can take a time linear in both the number of solutions of A and B.

The operators having such a non-linear behavior on unsafe terms are: joins, left joins, strict minuses and minuses. For each of them, the complexity of the approach in our prototype is in the product of their inputs (in the worst case). When $(p_A \cap p_B) \setminus (c_A \cap c_B)$ is very small (e.g. just one value C) we can also rewrite A and B (e.g. let $(X = A)$ in $\sigma_{bnd(c)}(X) \cup \sigma_{\neg bnd(c)}(X)$) but this rewriting is exponential in general.

7.2 Estimation of μ-algebra terms cardinality

Estimating the cardinality of queries is well studied subject. Been new, μ-algebra has not yet been at the heart of such works but most works on cardinality estimation for the relational algebra, for SQL, for SPARQL, for recursive queries, etc. do apply to our μ-algebra.

In our prototype we rely on simple estimation based on triple selectivity [5].

8 RELATED WORK

The optimized evaluation of SPARQL is well studied especially for the BGP fragment. The evaluation of recursive queries is also a well studied subject especially outside of the SPARQL language. In this section we propose to compare our approach to various lines of work that have tackled the subject from the more ad-hoc, tailored for SPARQL to very general approaches: Reachability joins, Waveguide, Regular Path queries, SQL extended for recursive queries and finally recursive datalog.

8.1 The relational algebra

The relational algebra introduced by Codd built the foundations of our work. Similarly to our work the relational algebra operates on a “set semantic” (while SQL operates on a “bag” semantic). The relational algebra differs from our work in several points. The two most salient ones being that the relational algebra works on a fixed domain and that it is not equipped with a fixpoint operator.

There are been works[1] to extend the relational algebra with an operator α representing recursive queries. If R is a relational algebra term with two attributes $\alpha(R)$ represents the transitive closure of R. If this operator is sufficient to represent SPARQL, it does not allow for a plan space as large as our μ-algebra (see Waveguide below).

8.2 SQL

SQL is based on the relational algebra. Both have been extensively studied either for themselves or in the context of SPARQL query evaluation. However using SQL for the optimization of SPARQL has been not been very successful even without considering PP [7, 12].

Even if there are not supported by all SQL stores, recursive queries do exist in SQL. Vendors introduced in their product various extensions allowing some kind of recursion and in its 99 version, SQL introduced Common Table Expressions (CTE) allowing recursive queries. Recursive CTE allow for very broad kind of recursive queries, broader than what is allowed in the alpha-extended. However not all SQL

databases support all these features (e.g. MySQL does not even support recursive CTE) and vendors generally consider CTE as "optimization fences". We benchmarked several SQL in our benchmark comparison (see section 9).

Postgres & SQLite. Postgres and SQLite both consider Common Table Expressions (CTE) as optimization barriers. It is thus not surprising that they actually computes the whole knows_star relation before selecting pairs (s, o) such that $s = name\_42$.

Our tests show this quadratic behavior (see fig 6). Even PSQL (with indexes on s and o for the table *knows*)

MySQL. As of today, MySQL does not support CTE.

8.3 Native sparql

8.3.1 ARQ. Jena is an Apache framework for building semantic web and linked data applications. ARQ is the SPARQL query evaluator of Jena and it has full SPARQL 1.1 support. Figure 6 reports on the time spent evaluating queries on two queries ARQ_1 and ARQ_2. ARQ_1 is our benchmark query while ARQ_2 is the same query but where we exchanged the order of the PP. As the figure shows ARQ sometimes also has the quadratic behaviour (depending on the order the PP in the input query).

8.3.2 Virtuoso. Virtuoso also optimize queries and is pretty fast (the fastest beside our prototype) but as we see in the close-up (right side of figure 6) it still has a quadratic behaviour. Virtuoso does not seem to depend on the order of PP in the query.

8.3.3 Waveguide. Waveguide[19] introduced Waveguide Plans (WGP) allowing the optimized evaluation of property paths. WGP mix together α-plans (which are plans based on the relational algebra) and FA-plans (which are based on automata). They show that their method subsumes both approaches, more precisely they show that both approaches force the associativity.

Given two terms φ and ψ, we define as syntactic sugar: $\varphi/\psi = \pi_{mid}\left((\rho_o^{mid}(\varphi)) \bowtie (\rho_s^{mid}(\psi))\right)$; given the regular expression $a/b/c$ the automata approach will compute the binary relation R recursively with $R_0 = \beta_s^o(N)$ and $R_{i+1} = ((R_i/a)/b)/c$ while the α-approach will compute $R_{i+1} = R_i/(a/b/c)$. In the mean time the WGP also consider the following computation: $R_{i+1} = (R_i/a)/(b/c)$. Notice that our approach also considers these three possible plans.

Given several PP, Waveguide translates each and then join them. In comparison, our approach translate each PP into a μ-algebra term and join them but before the optimization of the query. Which means the evaluation of PP can be interleaved. For instance given 3 PP: ?a *knows*$*$?b, ?b *lastname Doe* and ?b *firstname John* our optimizer propose the plan that starts by finding the valid nodes ?b then finds the ?a reachable from these ?b. Waveguide being not publicly available, it is not included in our benchmark.

8.3.4 Reachability Joins. Reachability joins are the basis of the Ferrari system [8] which brought fast Property Paths into RDF-3X. The idea is that given two terms A and B, both defining binary relation, their reachability join (noted $A \bowtie_R B$) is equal to the join of A with the transitive closure of B.

The Ferrari system is very fast for star over patterns for which it pre-computed reachability indexes but it incapable of handling the computing arbitrary stars (which can appear in SPARQL). Furthermore, from an expressiveness point of view, $A \bowtie_R B = A \bowtie \alpha(B)$ and therefore it suffers the same problem with associativity as *alpha*-plans: it cannot express plans such as $\mu(X = \beta_s^o(N) \cup (X/A)/(B/C))$.

8.4 Datalog

Finally, a major line of research to tackle recursive queries is datalog. The optimization and fast execution of datalog on graph data is a challenge due to the expressive power of datalog and its logic-based form [6]. In our benchmark we used the well-known *datalog*[4] tool and the *dlv* for Datalog evaluation.

9 BENCHMARK

In this section we report on our benchmark comparing our prototype with other query engines. Our benchmark is composed of a unique and very simple query because it exemplifies the lack of optimization in current query engines as noted by gMark [4]: our prototype is linear and all other tools have quadratic behaviour.

The graph we considered is a simple loop `<http://example/name_`i`>` is linked to `<http://example/name_`$i+1$`>`. Our main query is presented in figure 6.

We hand-translated our queries to SQL. The SQL queries had to be slightly adapted for each SQL store but we stored the data for each prefix with one table "subject-object" for each predicate. For our toy query the SQL version for Postgresql is:

```
WITH RECURSIVE knows_star AS (
      SELECT *  FROM knows
   UNION ALL
      SELECT knows_star.s AS s, knows.o AS o
      FROM knows_star, knows
      WHERE knows_star.o=knows.s
) SELECT * FROM knows_star k, named n
WHERE n.o='name_42' and k.o=n.s;
```

Depending on the order of the PP in the query there are two translation. Our translation is given below and only asks whether there is a solution. The other translation produces the same datalog but the third line is `sol :- path(42,_)`. In figure 6, DLV_1 and $datalog_1$ correspond to the direct translation while DLV_2 and $datalog_2$ to the second. All seem to take quadratic time yet the direct translation is slower than the reversed translation.

[4] http://www.ccs.neu.edu/home/ramsdell/tools/datalog/

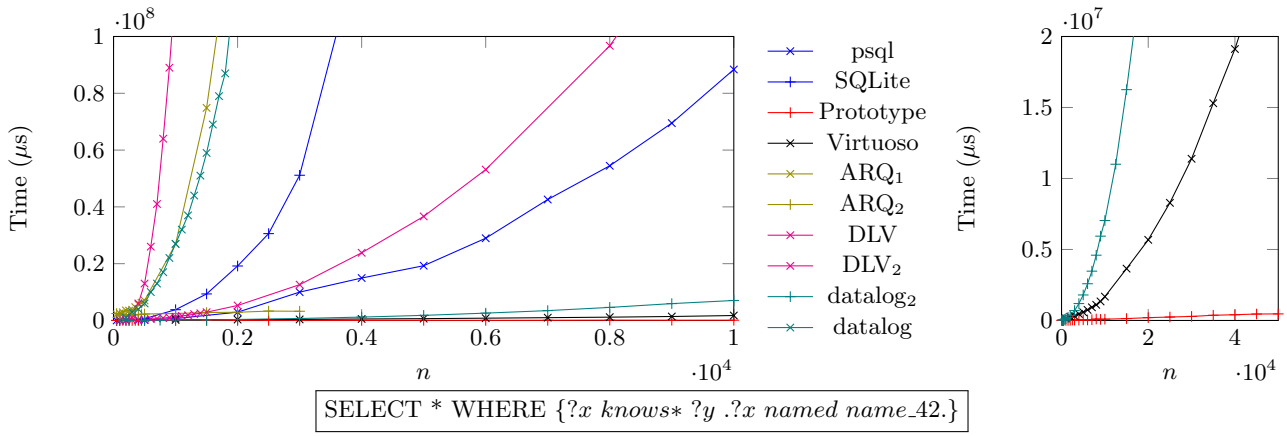

Figure 6: Evaluation time for the query on a graph of size n with a second framing for datalog$_2$, Virtuoso and our prototype

```
path(X,X) :- knows(X,_). path(X,X) :- knows(_,X).
path(X,Y) :- path(X,Z),knows(Z,Y).
sol :- path(_,42).
sol?
```

9.1 Our prototype

The query of our benchmark is the example query of section 4. We verify experimentally that the execution time for our prototype is linear.

10 CONCLUSION AND FUTURE WORKS

We introduced μ-algebra, a variant of the SPARQL algebra. We showed that terms our algebra can be rewritten to produce new semantically equivalent terms. If we consider these terms as Query Execution Plans then the set of considered QEPs is strictly larger than what was considered by previous techniques. Experimental results show that our approach yields performance improvements even for very simple queries.

We also believe that our algebra represents a step toward the ambitious goal of unifying various traits of the relational algebra with traits of NoSQL languages in a common framework (syntax, semantics, typing, rewriting schemes). Our algebra subsumes the SPARQL Algebra with a more general recursion. As a perspective for further work, we plan to investigate how our approach can be improved along several directions: finer-grained cardinality estimation, distributed implementations for evaluating terms of our algebra, and extensions for the compilation of query languages with other data models.

11 APPENDIX

We plain to release publicly the code publicly but for double-blind purposes, you can find a complete list of rewritings, along with their proofs and our prototype source at the anonymized address: http://bit.ly/2z5eki0

REFERENCES

[1] R. Agrawal. 1988. Alpha: an extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering* 14, 7 (July 1988), 879–885. https://doi.org/10.1109/32.42731

[2] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. 2009. Extending SPARQL with Regular Expression Patterns (for Querying RDF). *Web Semant.* 7, 2 (April 2009), 57–73. https://doi.org/10.1016/j.websem.2009.02.002

[3] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 2012. Counting Beyond a Yottabyte, or How SPARQL 1.1 Property Paths Will Prevent Adoption of the Standard. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12)*. ACM, New York, NY, USA, 629–638. https://doi.org/10.1145/2187836.2187922

[4] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George HL Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: schema-driven generation of graphs and queries. *IEEE Transactions on Knowledge and Data Engineering* 29, 4 (2017), 856–869.

[5] Abraham Bernstein, Christoph Kiefer, and Markus Stocker. 2007. *OptARQ: A SPARQL optimization approach based on triple pattern selectivity estimation.* University.

[6] S. Ceri, G. Gottlob, and L. Tanca. 1989. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering* 1, 1 (Mar 1989), 146–166. https://doi.org/10.1109/69.43410

[7] Orri Erling and Ivan Mikhailov. 2009. *RDF Support in the Virtuoso DBMS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 7–24. https://doi.org/10.1007/978-3-642-02184-8_2

[8] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. 2013. Sparqling Kleene: Fast Property Paths in RDF-3X. In *First International Workshop on Graph Data Management Experiences and Systems (GRADES '13)*. ACM, New York, NY, USA, Article 14, 7 pages. https://doi.org/10.1145/2484425.2484443

[9] Egor V. Kostylev, Juan L. Reutter, Miguel Romero, and Domagoj Vrgoč. 2015. SPARQL with Property Paths. In *The Semantic Web - ISWC 2015*, Marcelo Arenas, Oscar Corcho, Elena Simperl, Markus Strohmaier, Mathieu d'Aquin, Kavitha Srinivas, Paul Groth, Michel Dumontier, Jeff Heflin, Krishnaprasad Thirunarayan, Krishnaprasad Thirunarayan, and Steffen Staab (Eds.). Number 9366 in Lecture Notes in Computer Science. Springer International Publishing, 3–18. DOI: 10.1007/978-3-319-25007-6_1.

[10] Katja Losemann and Wim Martens. 2012. The Complexity of Evaluating Path Expressions in SPARQL. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '12)*. ACM, New York, NY, USA, 101–112. https://doi.org/10.1145/2213556.2213573

[11] Katja Losemann and Wim Martens. 2013. The Complexity of Regular Expressions and Property Paths in SPARQL. *ACM Trans. Database Syst.* 38, 4 (Dec. 2013), 24:1–24:39. https://doi.org/10.1145/2494529

[12] T. Neumann and G. Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *2011 IEEE 27th International Conference on Data Engineering*. 984–994. https://doi.org/10.1109/ICDE.2011.5767868

[13] Carlos Ordonez. 2010. Optimization of Linear Recursive Queries in SQL. *IEEE Transactions on Knowledge and Data Engineering* 22, 2 (Feb. 2010), 264–277. https://doi.org/10.1109/TKDE.2009.83

[14] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2006. Semantics and Complexity of SPARQL. *CoRR* abs/cs/0605124 (2006). http://arxiv.org/abs/cs/0605124

[15] Martin Przyjaciel-Zablocki, Alexander Schätzle, Thomas Hornung, and Georg Lausen. 2011. RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In *The Semantic Web: ESWC 2011 Workshops*, Raúl García-Castro, Dieter Fensel, and Grigoris Antoniou (Eds.). Number 7117 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 50–64. DOI: 10.1007/978-3-642-25953-1_5.

[16] Simon Schenk. 2007. *A SPARQL Semantics Based on Datalog*. Springer Berlin Heidelberg, Berlin, Heidelberg, 160–174. https://doi.org/10.1007/978-3-540-74565-5_14

[17] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory (ICDT '10)*. ACM, New York, NY, USA, 4–33. https://doi.org/10.1145/1804669.1804675

[18] Michael Schmidt, Michael Meier, and Georg Lausen. 2010. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory (ICDT '10)*. ACM, New York, NY, USA, 4–33. https://doi.org/10.1145/1804669.1804675

[19] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2015. Towards Query Optimization for SPARQL Property Paths. *arXiv:1504.08262 [cs]* (April 2015). http://arxiv.org/abs/1504.08262 arXiv: 1504.08262.