On the Optimization of Recursive Relational Queries: Application to Graph Queries

Louis Jachiet, Pierre Genevès, Nils Gesbert, Nabil Layaïda

On the Optimization of Recursive Relational Queries

Louis Jachiet
DI ENS, ENS, CNRS, PSL Research University &
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP*, LIG
Paris, France
*Institute of Engineering Univ. Grenoble Alpes
louis.jachiet@ens.fr

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

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

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

ABSTRACT

Graph databases have received a lot of attention recently as they are particularly useful in many applications such as social networks or for the semantic web. Various languages have emerged to query such graph databases. At the heart of many of those query languages, there is a construction to navigate through the graph which allows some form of recursion.

The relational model has benefited from a huge body of research in the last half century and that is why many graph databases either rely on (or have adopted the techniques of) relational based query engines. Since its introduction, the relational model has seen various attempts to extend it with recursion and it is now possible to use recursion in several SQL or Datalog based database systems. The optimization of recursive queries remains, however, a challenge.

In this paper, we introduce μ-RA, a variation of the Relational Algebra that allows for the expression of relational queries with recursion. μ-RA can express unions of conjunctive regular path queries as well as certain non-regular properties. We present its syntax, semantics and the rewriting rules we specifically devised to tackle the optimization of recursive queries. A prototype evaluator implementing these rewriting rules is shown to be more efficient than previous approaches.

ACM Reference Format:
Louis Jachiet, Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2018. On the Optimization of Recursive Relational Queries. In *Proceedings of Bases de données avancées (BDA'18)*. ACM, New York, NY, USA, 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The querying capability of languages to extract information can be greatly improved with the introduction of some form of recursion. In graph databases, such features are particularly useful, which is why graph languages often include constructions such as Regular Path Queries (RPQ) [13], Conjunctive RPQ and various further extensions of them such as Union of CRPQ (UCRPQ) [8, 9, 11, 22]. For instance, SPARQL recently [18] introduced Property Paths, PGQL has RPQs, and proposals for OpenCypher [15] include Path Patterns; all these constructions contain a recursive capability.

In SPARQL, for instance, Property Paths express non-trivial but crucial relations in RDF data. They correspond to recursive regular expressions particularly useful to navigate in deeply linked graph structures such as those found in social networks, life sciences and transportation networks. For example, the following SPARQL query retrieves the species of birds that can be found somewhere in Romania:

```
SELECT * {
  ?species :subTaxon+  :Bird .
  ?species :livesIn/:locatedIn+ :Romania . }
```

using property path expressions that describe possible relations between species, Bird and Romania.

Finally, recursive queries are also particularly interesting in ontology-based data integration where they can be used to query knowledge implicitly encoded with transitive relations in ontologies [5].

The efficient evaluation of queries with property paths is a key issue with the increasing amounts of RDF data available [1, 19]. However, queries with property paths are notoriously known to be much harder to evaluate than non-recursive ones [23, 27]. Even with modest amounts of data, the recent benchmarking work found in [7] notices that "*all tested systems either failed on the majority of these* [recursive] *queries or had to be manually terminated after unexpectedly long running times.*" A main difficulty of property path evaluation comes from the fact that it is not straightforward to find an appropriate plan for efficient evaluation.

Contribution. We introduce an extended relational model with a fixpoint operator that captures UCRPQ. We introduce rewrite rules for terms with fixpoint which allow generating new execution plans. We show empirically that a prototype implementation generating these new plans performs better than previous approaches.

BDA'18, Octobre 2018, Bucarest

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

Outline of the paper. The paper is decomposed as follows: Section 2 presents μ-RA: our variation of the relational algebra (RA) to equip it with a fixpoint construction inspired by the μ-calculus [21]. In Section 3, we show how recursive queries over graphs can be translated into μ-RA. Section 4 describes properties of μ-RA and demonstrates that our fixpoint can be rewritten (to push filters, projections and joins inside of the fixpoints) and that two (or more) fixpoints can sometimes be merged into a unique fixpoint. Section 5 explains why our approach is able to generate efficient plans that were beyond reach.

For reading purposes, we present only proof sketches of our main theorems; the full proofs are all available in the appendix available at https://louis.jachiet.com/tmp/mura.pdf

2 THE μ-EXTENDED REL. ALGEBRA

In this section we present our variation of the domain-independent relational algebra, equipped with a fixpoint, which we call μ-RA (μ-extended relational algebra). This section first recalls some usual definitions. Then we present our syntax, types and semantics that are adapted from the RA for our new construct.

2.1 Data model

Our data model is the same as for the classical relational algebra: we consider *relations* which are sets of *mappings* (also called tuples, or lines) which associate *column names* to *values*.

Formally, we assume the following constants:

- $\mathfrak{V}$ an infinite set of *values*;
- $\mathfrak{C}$ an infinite set of *column names*;
- $\mathfrak{R}$ an infinite set of *relation names*;

Definition 1. *A* mapping *or tuple is a partial function* $m : \mathfrak{C} \to \mathfrak{V}$ *whose domain is finite. If* $dom(m) = \{c_1, \ldots, c_n\}$, m *can also be seen as the set* $\{c_1 \to m(c_1), \ldots, c_n \to m(c_n)\}$.

Definition 2. *Two mappings* m_1 *and* m_2 *are* compatible*, noted* $m_1 \sim m_2$*, when* $\forall c \in dom(m_1) \cap dom(m_2),\ m_1(c) = m_2(c)$*. If* m_1 *and* m_2 *are compatible, we define* $m_1 + m_2 : dom(m_1) \cup dom(m_2) \to \mathfrak{V}$ *by:*

$$(m_1 + m_2)(c) = \begin{cases} m_1(c) & \text{if } c \in dom(m_1) \\ m_2(c) & \text{if } c \in dom(m_2) \end{cases}$$

If we see mappings as sets, this corresponds to their union.

Definition 3. *A* relation *is a finite set of mappings which share the same domain. We call this common domain the* type *of the relation. Note that we do not consider datatypes in this paper (all values are in the single domain* $\mathfrak{V}$*): a type is just a set of column names.*[1]

The empty relation is considered compatible with all types.

Relations represent data. A relational database is a finite set of named relations (also called tables). We represent such a database as a triple $(\mathcal{R}, \Gamma, D)$ where: $\mathcal{R} \subset \mathfrak{R}$ is the set of relation names; Γ, the database schema, associates relation names to relation types; and D, the database body, associates relation names to actual relations. The body must be consistent with the schema, i. e. for any $R \in \mathcal{R}$, $D(R)$ has type $\Gamma(R)$.

[1] We do this for simplicity; datatypes could be added by replacing column names with pairs of a column name and a datatype without changing our theory much.

2.2 Syntax of μ-RA terms

Our algebra μ-RA is mainly a variation of the relational algebra, with the addition of a fixpoint operator μ; it operates on relations. The terms represent queries and are built from relation variables and operations; given a mapping from variables to relations (representing a database body), a term can be evaluated and yields another relation (the solution of the query). Evaluation of terms is described in Section 2.3.

2.2.1 Filters. The standard selection operation $\sigma_{\mathfrak{f}}$, which operates on a relation by keeping only a subset of its mappings, depends on a *filter* $\mathfrak{f}$ indicating which mappings are to be kept. This filter can be seen as a function from mappings to booleans.

To keep things focused, we do not detail here a syntax for filters, but we assume that for any filter $\mathfrak{f}$ we can compute a set $FC(\mathfrak{f})$ of column names such that the result of $\mathfrak{f}(m)$ depends only on $\{m(c) \mid c \in FC(\mathfrak{f})\}$.

2.2.2 Terms. The core syntax of terms is defined in Figure 3. The base terms are relation variables X and constants $|c \to v|$ (representing a single mapping with a singleton domain). Two relations can be combined with the classical relational operators $\cup$, $\bowtie$ and $\triangleright$. One relation can be filtered using the classical selection operation $\sigma_{\mathfrak{f}}$ where $\mathfrak{f}$ is a filter.

Less classically, the unary operation $\beta_a^b(\cdot)$ (column duplication) copies the values of column a onto column b, and the anti-projection $\widetilde{\pi}_a(\cdot)$ (or column dropping) removes column a. These operations can express the classical operations projection and renaming:

- The rename operator $\rho_a^b(\varphi)$ (renaming the column a into b) is defined as syntactic sugar for $\widetilde{\pi}_a\left(\beta_a^b(\varphi)\right)$;
- The projection operator $\pi_{p_1,\ldots,p_n}(\varphi)$ can be expressed in terms of $\widetilde{\pi}(\cdot)$ provided we know the type of φ: if φ has type $t = \{p_1, \ldots, p_n, a_1, \ldots, a_k\}$ we have $\pi_{p_1,\ldots,p_n}(\varphi)$ equivalent to $\widetilde{\pi}_{a_1}\left(\ldots \widetilde{\pi}_{a_k}(\varphi)\right)$. Our choice of anti-projection will allow us to extend the domains of subterms without changing the projections, as in $\widetilde{\pi}_a(\varphi) \bowtie \psi \to \widetilde{\pi}_a(\varphi \bowtie \psi)$ when a is not in the type of ψ.

Finally, we introduce the fixpoint term $\mu(X = \varphi)$ representing a recursive query. In this term, there are some additional restrictions on φ, which will be detailed in Section 2.5. The result R of this operation is a fixpoint in the sense that evaluating φ with X bound to R must yield R again. The restrictions we add ensure that this fixpoint exists and can be computed iteratively.

We consider μ as a variable binder, yielding the standard notions of *free* and *bound* variable occurrences:

Definition 4. *In a term* φ*, all occurrences of a variable* X *which appear in a subterm of the form* $\mu(X = \psi)$ *are* bound*. All other occurrences of* X *are* free*.*

As will be clear from the semantics, bound variables can be renamed, as usual, without changing the meanings of the terms. We can thus assume for simplicity that all bound variables are different from each other and from free variables.

2.3 Semantics

In μ-RA, relation variables X are used to denote both references to a database relation and a recursive relation. In a full query, the

$$
\begin{array}{rcl}
[\![\varphi_1 \bowtie \varphi_2]\!]_V & = & \{m_1 + m_2 \mid m_1 \in [\![\varphi_1]\!]_V \wedge m_2 \in [\![\varphi_2]\!]_V \wedge m_1 \sim m_2\} \\
[\![\varphi_1 \triangleright \varphi_2]\!]_V & = & \{m \in [\![\varphi_1]\!]_V \mid \forall m' \in [\![\varphi_2]\!]_V \ \neg(m' \sim m)\} \\
[\![\widetilde{\pi}_a(\varphi)]\!]_V & = & \left\{\{c \to v \in m \mid c \neq a\} \,\middle|\, m \in [\![\varphi]\!]_V\right\} \\
[\![\beta_a^b(\varphi)]\!]_V & = & \left\{\{c \to v \in m \mid c \neq b\} \cup \{b \to v \mid a \to v \in m\} \,\middle|\, m \in [\![\varphi]\!]_V\right\} \\
[\![\mu(X = \varphi)]\!]_V & = & [\![X]\!]_{V[X/U_\infty]} \quad \text{where } U_0 = \emptyset,\ U_{i+1} = U_i \cup [\![\varphi]\!]_{V[X/U_i]}, \text{ and } U_\infty = \bigcup_{n \in \mathbb{N}} U_i
\end{array}
\qquad
\begin{array}{rcl}
[\![\varphi_1 \cup \varphi_2]\!]_V & = & [\![\varphi_1]\!]_V \cup [\![\varphi_2]\!]_V \\
[\![|c \to v|]\!]_V & = & \{\{c \to v\}\} \\
[\![X]\!]_V & = & V(X) \\
[\![\sigma_{\mathfrak{f}}(\varphi)]\!]_V & = & \{m \mid m \in [\![\varphi]\!]_V \wedge \mathfrak{f}(m) = \top\}
\end{array}
$$

Figure 1: Semantics of μ-RA

$$
\Gamma \vdash |c \to v| : c \qquad \frac{\Gamma(X) = t}{\Gamma \vdash X : t} \qquad \frac{\Gamma \vdash \varphi_1 : t \qquad \Gamma \vdash \varphi_2 : t}{\Gamma \vdash \varphi_1 \cup \varphi_2 : t} \qquad \frac{\Gamma \vdash \varphi_1 : t_1 \qquad \Gamma \vdash \varphi_2 : t_2}{\Gamma \vdash \varphi_1 \bowtie \varphi_2 : t_1 \cup t_2} \qquad \frac{\Gamma \vdash \varphi_1 : t_1 \qquad \Gamma \vdash \varphi_2 : \_}{\Gamma \vdash \varphi_1 \triangleright \varphi_2 : t_1}
$$

$$
\frac{\Gamma \vdash \varphi : t \qquad FC(\mathfrak{f}) \subseteq t}{\Gamma \vdash \sigma_{\mathfrak{f}}(\varphi) : t} \qquad \frac{\Gamma \vdash \varphi : t \qquad a \in t \qquad b \notin t}{\Gamma \vdash \beta_a^b(\varphi) : t \cup \{b\}} \qquad \frac{\Gamma \vdash \varphi : t \qquad a \in t}{\Gamma \vdash \widetilde{\pi}_a(\varphi) : t \setminus \{a\}} \qquad \frac{\Gamma \cup \{X \to t\} \vdash \varphi : t}{\Gamma \vdash \mu(X = \varphi) : t}
$$

Figure 2: Typing rules for μ-RA

φ	::=		term
		X	relation variable
	|	$\|c \to v\|$	constant
	|	$\varphi_1 \cup \varphi_2$	union
	|	$\varphi_1 \bowtie \varphi_2$	join
	|	$\varphi_1 \triangleright \varphi_2$	antijoin
	|	$\sigma_{\mathfrak{f}}(\varphi)$	filtering
	|	$\beta_a^b(\varphi)$	duplication
	|	$\widetilde{\pi}_a(\varphi)$	anti-projection
	|	$\mu(X = \varphi)$	fixpoint

Figure 3: Grammar of μ-RA

two are distinguished by the fact that database references appear as free variables, whereas recursion variables are bound by μ; but in a subterm, we do not need to distinguish the two. In all cases, the semantics of a term φ depends on an *environment* V which maps all free variables of φ to relations.

The semantics is defined in Figure 1, where $[\![\varphi]\!]_V$ designates the result of evaluating φ in the environment V. This result is defined recursively from the results of evaluating the subterms. The initial environment for evaluating the whole term is a database body D, but in evaluating $\mu(X = \varphi)$, the recursive calls use different environments where the recursion variable X is given a value: the notation $V[X/S]$ represents the environment V altered by mapping X to S.

2.4 Type System

Given a schema Γ for a set of relation variables $\mathcal{R}$, we can infer types for terms whose free variables are in $\mathcal{R}$. The typing judgement $\Gamma \vdash \varphi : t$ means that when evaluated in an environment conforming to the schema Γ, φ will yield a relation of type t.

Our type system is defined by the rules on Figure 2; it is quite straightforward. The only difficulty is to infer a type for fixpoints (last rule), since the rule does not give an explicit way to guess the value of t. However, this rule is still operational. Indeed, we can start typing φ with an unknown type for X (a type variable). During the typing, the value of this type variable can get constrained (typically, if $\cup$ is used it forces the two types to be equal). Then when we finish there are three possibilities: either t does not exist (the constraints are incompatible with each other), meaning the whole term is not typable; or t is entirely determined and we computed it; or the constraints are not enough to determine t entirely, meaning that the term is actually typable with different values for t. In the last case, it means that the result of the fixpoint term is always empty. Indeed, the type of a relation is the common domain of all its mappings; it is unique unless the relation is empty.

Given a database schema Γ, we write $\mathcal{F}[\Gamma]$ for the set of well-typed terms in Γ (i. e. the terms φ such that $\Gamma \vdash \varphi : t$ holds for some t).

Proposition 1. *Given a database $(\mathcal{R}, \Gamma, D)$ and $\varphi \in \mathcal{F}[\Gamma]$, if $\Gamma \vdash \varphi : t$ then the relation $[\![\varphi]\!]_D$ has type t.*

Example 1. *Let us suppose that we want to compute the transitive closure of R of type $\{a, b\}$. The closure is captured by the term $\mu\left(X = R \cup \widetilde{\pi}_m\left(\rho_b^m(R) \bowtie \rho_a^m(X)\right)\right)$ of type $\{a, b\}$.*

Indeed, the fixpoint should have the type of its constant part which is R of type $\{a, b\}$. Then we can check the type of the non-constant part: $\rho_b^m(R)$ has type $\{a, m\}$ and is joined with $\rho_a^m(X)$ of type $\{b, m\}$. The result has type $\{a, b, m\}$ but the m column is discarded by the $\widetilde{\pi}_m(\dots)$.

2.5 Restrictions on fixpoints

As we will show in this section, our syntax for the μ-RA is very general and comprise some counter-intuitive fixpoints. In this section we present restrictions on fixpoints that are needed for the validity of our rewrite rules and of most our propositions and theorems.

After this section, we will suppose that all fixpoints abide the restrictions that we present here. This does not mean, however, that

our method can not be applied on general terms: given a general term φ that contains a subterm ψ, if ψ abides the restrictions then we can apply our rewrite rules on ψ.

2.5.1 Inflationary semantics for fixpoints. Fixpoints in the relational algebra have been considered under several semantics. One of the main split regarding the semantics of fixpoints is whether the fixpoint semantics is *inflationary* or *non inflationary*.

In the *inflationary* semantics, the fixpoint is evaluated as the limit of a sequence of the form $U_{i+1} = U_i \cup [\![\varphi]\!]_{V[X/U_i]}$ while in a non-inflationary semantics the fixpoint is evaluated as the limit of $U_{i+1} = [\![\varphi]\!]_{V[X/U_i]}$ (such a limit might not exist).

Our semantics is inflationary, which allows all terms to have a defined meaning, although this meaning is not always intuitive. For example, the syntax as defined by our grammar allows $\varphi = \mu(X = |c \to 0| \triangleright X)$. The inflationary semantics of this term is $\{\{c \to 0\}\}$ (because each U_i contains the previous one). However it is not, in this case, a fixpoint semantics: $[\![\varphi]\!]_{V[X/\{\{c\to 0\}\}} = \emptyset$. Usual transformations like unfolding a fixpoint are thus not possible in general on all terms.

As we will see in proposition 2, on our restricted terms the inflationary and non-inflationary semantics coincide, which means that the result of this paper could also be used in a recursive relational algebra with a non-inflationary semantics.

2.5.2 Properties of fixpoints.

Definition 5. *Given a term φ, we say that φ is* constant *in X when X is not a free variable of φ.*

Definition 6. *A fixpoint term $\mu(X = \varphi)$ is said:*

- positive *when for all subterms $\varphi_1 \triangleright \varphi_2$ of φ, φ_2 is constant in X;*
- linear *when for all subterms of φ of the form $\varphi_1 \bowtie \varphi_2$ or $\varphi_1 \triangleright \varphi_2$, either φ_1 or φ_2 is constant in X;*
- mutually recursive *when there exists a subterm $\mu(Y = \psi)$ of φ with X free in ψ.*

Proposition 2. *If $\mu(X = \varphi)$ of type t is linear, positive and non mutually recursive then the function $f(S) = [\![\varphi]\!]_{V[X/S]}$ (for S a set of mappings of type t) is such that:*

$$f(S) = f(\emptyset) \cup \bigcup_{x \in S} f(\{x\})$$

and thus f has a fixpoint with $[\![\mu(X = \varphi)]\!]_V = f^\infty(\emptyset)$.

Proof sketch. We prove by induction on the subterms ξ of φ where X is free in ξ that $[\![\xi]\!]_{V[X/S]} = [\![\xi]\!]_{V[X/\emptyset]} \cup \bigcup_{x \in S} [\![\xi]\!]_{V[X/\{x\}]}$. By linearity, such a ξ can only be combined with a constant term and by positivity, it cannot be negated. □

2.5.3 Expressivity of restricted fixpoints. These restrictions do have an effect on the expressivity of our language. We can show (see appendix B) that μ-RA has, at least, the expressive power of inflationary-Datalog$^\neg$ (Datalog with inflationary semantics and negation). When we restrict fixpoints to be positive and non mutually recursive then our language has exactly the expressive power of stratified-Datalog$^\neg$. Finally with all our restrictions, our language has exactly the expressive power of linear datalog which is shown to be strictly less expressive than stratified-Datalog$^\neg$ [?].

This fragment, however, does contain a lot of interesting queries, for instance the next section present how to translate UCRPQ into μ-RA (with the restrictions).

Furthermore, as we explained at the beginning of this section, our method can be applied on general terms. However, for the sake of simplicity, in the rest of this paper we will only consider the fragment *rest-μ-RA* of μ-RA containing only terms where all the fixpoints are **linear, positive and non mutually recursive**.

2.6 Decomposed fixpoints

Once our terms are restricted, we see that fixpoints can actually be decomposed into a strictly *recursive* part and a *constant* part. This decomposition will be later useful into expressing some of our rewrite rules.

Definition 7. *Given a term φ linear and positive in X, we say that φ is recursive in X when $rec(\varphi, X) = \top$ with rec defined as:*

$$\begin{aligned}
rec(\varphi_1 \cup \varphi_2, X) &= rec(\varphi_1, X) \wedge rec(\varphi_2, X)\\
rec(\varphi_1 \bowtie \varphi_2, X) &= rec(\varphi_1, X) \vee rec(\varphi_2, X)\\
rec(\varphi_1 \triangleright \varphi_2, X) &= rec(\varphi_1, X)\\
rec(\sigma_{\mathfrak{f}}(\varphi), X) &= rec(\varphi, X)\\
rec(\widetilde{\pi}_a(\varphi), X) &= rec(\varphi, X)\\
rec(\beta^b_a(\varphi), X) &= rec(\varphi, X)\\
rec(\mu(Y = \varphi), X) &= \bot\\
rec(X, Y) &= X = Y\\
rec(|c \to v|, X) &= \bot
\end{aligned}$$

Being recursive or constant are syntactical properties. However the two following propositions give a semantic interpretation of those syntactical properties.

Lemma 1. *Let φ be a term.*

- *If φ is recursive in X then for all V, $[\![\varphi]\!]_{V[X/\emptyset]} = \emptyset$.*
- *If φ is constant in X, then φ does not depend on X, i.e. for all S and V, $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\emptyset]}$.*

Definition 8. *A fixpoint term $\mu(X = \kappa \cup \psi)$ is said* decomposed *when κ is constant in X and ψ is recursive in X.*

Example 2. *The term $\mu\left(X = R \cup \widetilde{\pi}_m\left(\rho^m_b(R) \bowtie \rho^m_a(X)\right)\right)$ of Example 1 is a* decomposed fixpoint*: R is constant and $\widetilde{\pi}_m\left(\rho^m_b(R) \bowtie \rho^m_a(X)\right)$ is recursive in X.*

Proposition 3. *A fixpoint term $\mu(X = \varphi)$, linear, positive and non mutually recursive can be rewritten to either: an empty term, a term φ with one less fixpoint or a decomposed fixpoint.*

3 QUERIES OVER GRAPHS

An important use case for recursive queries is graph databases. Although our algebra is based on the relational model, it is able to model queries over graphs. Indeed, consider a directed graph where edges are labelled. We can assume that both vertices and edge labels are elements of our set of values $\mathfrak{V}$.

The graph can then be represented as a pair $(\mathcal{V}, \mathcal{E})$ with $\mathcal{V} \subset \mathfrak{V}$ and $\mathcal{E} \subset \mathcal{V} \times \mathfrak{V} \times \mathcal{V}$. This can be modelled as a relational database with two relations V and E representing these two sets, with the schema $\Gamma = \{V \to \{f\}, E \to \{f, l, t\}\}$ where f, l, t stand

respectively for *from*, *label*, *to* (the column name f for V is arbitrary but will avoid some renamings).

As we discussed in the introduction, there are many formalisms for expressing recursive queries on graphs, most of which include a language of regular expressions to describe a set of paths (regular path queries or RPQ). As a simple example, take the following syntax for regular path expressions r:

r	::=	
	v	a single label
|	r_1/r_2	concatenation
|	$r_1 \vert r_2$	Alternative
|	r^{-1}	Reverse
|	$r?$	Zero or One
|	r^*	Kleene star
|	r^+	Transitive closure

These regular expressions denote sequences (words) of labels in the standard way. We want to translate them to queries such that the result of evaluating the query $(\!|r|\!)$ on our graph database is the set of all mappings $\{f \to v_1, t \to v_2\}$ such that there is a path from v_1 to v_2 in the graph whose sequence of labels matches r.

One possible translation is the following:

$$
\begin{aligned}
(\!|v|\!) &= \widetilde{\pi}_l\left(\sigma_{l=v}\left(E\right)\right)\\
(\!|r_1/r_2|\!) &= \widetilde{\pi}_m\left(\rho_t^m\left((\!|r_1|\!)\right) \bowtie \rho_f^m\left((\!|r_2|\!)\right)\right)\\
(\!|r_1|r_2|\!) &= (\!|r_1|\!) \cup (\!|r_2|\!)\\
(\!|r^{-1}|\!) &= \rho_m^f\left(\rho_f^t\left(\rho_t^m\left((\!|r|\!)\right)\right)\right)\\
(\!|r?|\!) &= \beta_f^t\left(V\right) \cup (\!|r|\!)\\
(\!|r^+|\!) &= \mu\Big(X = (\!|r|\!) \cup \widetilde{\pi}_m\left(\rho_t^m\left((\!|r|\!)\right) \bowtie \rho_f^m\left(X\right)\right)\Big)\\
(\!|r^*|\!) &= \mu\Big(X = \beta_f^t\left(V\right) \cup \widetilde{\pi}_m\left(\rho_t^m\left((\!|r|\!)\right) \bowtie \rho_f^m\left(X\right)\right)\Big)
\end{aligned}
$$

Note that such a translation produces some V symbols that might have been eliminated in another translation. For instance on $:A/:B?$ where $:A$ and $:B$ are labels, our translation will produce a V to handle the case of paths where $:B$ is not used. Another translation could decide to treat this special pattern as the equivalent pattern $:A|:A/:B$ that does not require to get the whole set V of nodes. Note that this rewriting is not always optimal as on regular expressions such as $(:A/:B?)/:C$ it chooses between $(:A|:A/:B)/:C$ while $:A/(:C|:B/:C)$ might be more efficient to compute. In our translation, we translate $r?$ and $r*$ "naively" and thus we introduce V symbols that could have been eliminated. However, we can then introduce a rewrite rule $V \bowtie \varphi \to \varphi$ for all terms φ such that the type of φ contains $\{f\}$. This way, we leave the removal of V symbols to this optimizer (this is what we actually implemented in our prototype described in section 5).

This translation of regular path expressions gives us the main brick to translate many graph query languages. With the relational operators, it is easy to extend this to conjunctive regular path queries and union of conjunctive regular path queries (UCRPQ); we can also represent SPARQL basic graph patterns composed of property paths (when interpreted in set semantics).

Note that the *rest-μ*-RA can also represent some queries which are not regular. For instance, let R be a binary relation between a and b, the term $\mu\left(X = \rho_b^p\left(R\right) \bowtie \rho_a^p\left(R\right) \cup \widetilde{\pi}_{a'}\left(\widetilde{\pi}_{b'}\left(\rho_a^{a'}\left(\rho_b^{b'}\left(X\right)\right) \bowtie \rho_a^{a'}\left(R\right) \bowtie \rho_b^{b'}\left(R\right)\right)\right)\right)$ computes the triples a, p, b such that there are a path from a to p of size $k > 0$ and from p to b of the same size k, which is not a regular property. This cannot be expressed in just UCRPQ, although it could be expressed in ECRPQ (extended conjunctive regular path queries [9]).

4 PROPERTIES OF THE REST-μ-RA

4.1 Motivation

The traditional RA has rewrite rules and the optimization of RA queries is usually done by rewriting to a (estimated) more efficient term using rewrite rules. In this section, we discuss properties of *rest-μ*-RA which allow us to introduce new rewrite rules, specific to terms with fixpoints. These rules are an addition to the classical rewrite rules of the RA, which are all valid on *rest-μ*-RA as well.

We first describe our four new rewrite rules informally; then in the following subsections we discuss the conditions under which these rules are valid. In Section 5, we will then see that our *rest-μ*-RA has plans reachable using our rewrite rules that other methods do not have.

4.1.1 Pushing filters into fixpoints. $\sigma_{\mathfrak{f}}\left(\mu(X = \varphi)\right) \to \mu\big(X = \sigma_{\mathfrak{f}}\left(\varphi\right)\big)$ (see Theorem 1). This always reduces the amount of mappings manipulated by the fixpoint. This rule is typically useful on *e.g.* RPQs such as $R^+(a, b)$ where a is a constant to force the evaluation to only compute the (a, b) where b is reachable from this a and not compute the whole R^+ before filtering the pairs with the wrong a.

4.1.2 Pushing joins into fixpoints. $\mu(X = \varphi) \bowtie \psi \to \mu(X = \varphi \bowtie \psi)$ (see Theorem 3). This can also lower the number of mappings solutions of the fixpoint. This rewrite rule can be used on RPQs such as R_1^+/R_2 where the naive translation would compute the whole relations R_1^+ and R_2 before joining them. With this rewrite rule, it would start from R_1/R_2 and at each iteration prepend a R_1. This would be typically useful when R_1/R_2 is smaller than just R_1 (e.g. the right side of R_2 might be a constant).

4.1.3 Merging fixpoints. $\mu(X = \varphi \cup \psi(X)) \bowtie \mu(X = \kappa \cup \xi(X)) \to \mu(X = \varphi \bowtie \kappa \cup \psi(X) \cup \xi(X))$ (see Theorem 4). This limits the number of fixpoints but also reduces the amount of mappings. This rewrite rule can be used on RPQs of the form R_1^+/R_2^+ similarly to the rule above: we compute R_1/R_2 and then at each step we either prepend R_1 or append R_2. This rule is also useful at the UCRPQ level, for instance to compute the combination of $R_1^+(a, b)$,$R_2^+(a, c)$,$R_3^+(a, d)$. By applying our rewrite rule twice, we obtain a term that computes $R_1(a, b)$, $R_2(a, c)$, $R_3(a, d)$ and at each iteration replace in the quadruplet (a, b, c, d) either b with b' (with $R_1(b, b')$) or c with c' (with $R_2(c, c')$), or d with d' (with $R_3(d, d')$).

4.1.4 Pushing antiprojections into fixpoints. $\widetilde{\pi}_p\left(\mu(X = \varphi)\right) \to \mu\big(X = \widetilde{\pi}_p\left(\varphi\right)\big)$ (see Theorem 5). This can reduce the number of mappings since as the value of the removed column is ignored, several mappings might get merged. For instance, on RPQs of the form (R_1^+/R_2) where the right value is discarded.

We now discuss formally the conditions which allow these rewritings. All the proofs are in Appendix A.

4.2 Image of a variable through a term

Given a term φ linear and positive in a variable X, we can compute a set of derivations from X for φ. And if w is a mapping then each mapping of $[\![\varphi]\!]_{V[X/\{w\}]}$ either belongs to $[\![\varphi]\!]_{V[X/\emptyset]}$ or is obtained using one of those derivations.

If we do not have $\sigma_{\mathfrak{f}}(\mu(X=\varphi)) \equiv \mu\big(X=\sigma_{\mathfrak{f}}(\varphi)\big)$ in general, it is because some mappings solution of $\mu(X=\varphi)$ might not pass the filter but still be useful to create mappings (with the fixpoint iteration) passing the filter condition. The study of those derivations will allow us to infer that e.g., in some circumstances, not passing the filter condition is something that will stay invariant by one iteration of the fixpoint, in which case we will have $\sigma_{\mathfrak{f}}(\mu(X=\varphi)) = \mu\big(X=\sigma_{\mathfrak{f}}(\varphi)\big)$.

4.2.1 Logical framework.

Definition 9. *The set of derivations $d(\varphi, X)$ is:*

$$
\begin{array}{lcl}
d(\varphi_1 \cup \varphi_2, X) & = & d(\varphi_1, X) \cup d(\varphi_2, X) \\
d(\varphi_1 \triangleright \varphi_2, X) & = & d(\varphi_1, X) \\
d(\varphi_1 \bowtie \varphi_2, X) & = & d(\varphi_1, X) \cup d(\varphi_2, X) \\
d(\widetilde{\pi}_a(\varphi), X) & = & \{p \circ (a \to \bot) \mid p \in d(\varphi, X)\} \\
d(\beta_a^b(\varphi), X) & = & \{p \circ (b \to a) \mid p \in d(\varphi, X)\} \\
d(\sigma_{\mathfrak{f}}(\varphi), X) & = & d(\varphi, X) \\
d(\mu(Y=\varphi), X) & = & \emptyset \\
d(X, X) & = & \{()\} \ \textit{(a singleton identity)} \\
d(X, R) & = & \emptyset \\
d(|c \to v|, X) & = & \emptyset
\end{array}
$$

Where $\circ$ represents the composition and $(a_1 \to b_1, \ldots, a_n \to b_n)$ represents the function that maps each a_i to its b_i and every other column name to itself. Note that this definition manipulates functions with an infinite domain but the domain where they do not coincide with the identity is finite and they are thus computable.

Example 1 followup. In our previous example, X appears only once and thus there is only one derivation that maps $a \to \bot, m \to \bot$ and everything else to itself. In particular b is mapped to itself.

Lemma 2. *Let w be a mapping and φ a term linear, positive and non mutually recursive in X. For all $m \in [\![\varphi]\!]_{V[X/\{w\}]}$ either $m \in [\![\varphi]\!]_{V[X/\emptyset]}$ or there exists $p \in d(\varphi, X)$ such that for all $c \in dom(w)$:*

$$\Big(p(c) = \bot\Big) \vee \Big(p(c) \notin dom(w)\Big) \vee \Big(m(c) = w\big(p(c)\big)\Big)$$

Definition 10. *Given a term φ linear and positive in a variable X, we define the* stabilizer *of X in φ as the following set of column names: $stab(\varphi, X) = \{c \in \mathfrak{C} \mid \forall p \in d(\varphi, X)\ p(c) = c\}$*

Lemma 3. *Given a fixpoint term $\mu(X=\varphi) \in \mathcal{F}[\Gamma]$ of type t and a mapping of type t, $w \in [\![\mu(X=\varphi)]\!]_V$ if and only if we can find a* lineage *$w_0, \ldots, w_n$ for w, that is $w_0, \ldots, w_n$ such that $w_0 \in [\![\varphi]\!]_{V[X/\emptyset]}$ and $w_{i+1} \in [\![\varphi]\!]_{V[X/\{w_i\}]}$.*

Furthermore for all lineages $w_0, \ldots, w_n$ and all $c \in t \cap stab(\varphi, X)$, we have $w_0(c) = w(c)$.

4.2.2 Application to rewrite rules.

Theorem 1 (Pushing filters). *Let $\mu(X=\varphi)$ be a fixpoint term, V an environment and f a filter condition with $FC(f) \subseteq stab(\varphi, X)$. Then we have $[\![\sigma_{\mathfrak{f}}(\mu(X=\varphi))]\!]_V = [\![\mu\big(X=\sigma_{\mathfrak{f}}(\varphi)\big)]\!]_V$, and if $\mu(X=\varphi)$ can be decomposed into $\mu(X=\kappa\cup\psi)$, we also have $[\![\sigma_{\mathfrak{f}}(\mu(X=\kappa\cup\psi))]\!]_V = [\![\mu\big(X=\sigma_{\mathfrak{f}}(\kappa)\cup\psi\big)]\!]_V$.*

Proof sketch. This is a consequence of lemma 3: we can filter the lineage on w or on w_0 but they have equal values on $FC(f)$ and by definition of FC, $eval(f, w_0) = eval(f, w)$. □

Theorem 2 (Pushing anti-joins). *Let $\mu(X=\varphi)$ be a fixpoint term, V an environment and ψ a term of type $t \subseteq stab(\varphi, X)$ (we suppose that X is not a free variable of ψ). Then we have $[\![\mu(X=\varphi) \triangleright \psi]\!]_V = [\![\mu(X=\varphi \triangleright \psi)]\!]_V$, and if $\mu(X=\varphi)$ can be decomposed into $\mu(X=\kappa\cup\xi)$, we also have $[\![\mu(X=\kappa\cup\xi) \triangleright \psi]\!]_V = [\![\mu(X=\kappa \triangleright \psi \cup \xi)]\!]_V$.*

Proof sketch. The anti join will act in very similar way to a filter since ψ is constant in X. Lineages $w_0 \ldots w_n$ will preserve the property to be compatible with one of the elements of $[\![\psi]\!]_V$. □

4.3 Reversing fixpoints

Our fixpoints often have a "direction". For instance, in the translation of r^+ that we presented, the term produced computes at the first iteration the pairs (f, t) such that $f \xrightarrow{r} t$ is an edge of the graph then it iteratively replaces (f, t) with (f, t') where $t \xrightarrow{r} t'$ is an edge of the graph. We could have designed the fixpoint to go the other way and replace at each step (f, t) with (f', t) such that $f' \xrightarrow{r} f$ is an edge of the graph.

The following proposition shows how to reverse the direction of fixpoints that are similar to the one that our translation produces for regular expressions of the form r^+ or r^*. It is possible to reverse fixpoints that have a much more general form as the examples of appendix E show.

Proposition 4. *Given a decomposed fixpoint of type $\{a, b\}$ of the form $\mu\big(X = \kappa \cup \widetilde{\pi}_c\left(\kappa \bowtie \rho_a^c(X)\right)\big)$, then it is equivalent to $\mu\Big(X = \kappa \cup \widetilde{\pi}_c\left(\rho_a^c\left(\rho_b^c(\kappa)\right) \bowtie \rho_b^c(X)\right)\Big)$.*

Example 1 followup. Building on our example, if we wanted to filter the transitive closure of R to limit b we could. If we wanted to filter on a we would first have to reverse the fixpoint using the proposition above.

4.4 Adding columns to fixpoints

In section 4.2 we have seen that the set of derivation describe what columns stay unchanged after one iteration of the fixpoints. In this section, we will study what happens when we change the type of a fixpoint by, e.g. adding or removing one column.

4.4.1 Logical framework.

Definition 11. *We say that a column $c \in \mathfrak{C}$ can be* added *to a fixpoint $\mu(X=\varphi) \in \mathcal{F}[\Gamma]$ when $add(\varphi, X, c) = \top$ holds, with add*

defined as:

$$\begin{array}{lcl} add(\varphi_1 \cup \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\ add(\varphi_1 \bowtie \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\ add(\varphi_1 \triangleright \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\ add(\widetilde{\pi}_a(\varphi), X, c) & = & add(\varphi, X, c) \text{ when } c \neq a \\ add(\widetilde{\pi}_c(\varphi), X, c) & = & X \notin free(\varphi) \\ add(\beta_a^b(\varphi), X, c) & = & add(\varphi, X, c) \wedge c \notin \{a, b\} \\ add(\sigma_f(\varphi), X, c) & = & add(\varphi, X, c) \wedge c \notin FC(f) \\ add(\mu(Y = \varphi), X, c) & = & add(\varphi, X, c) \\ add(R, X, c) & = & c \notin \Gamma(R) \text{ when } X \neq R \\ add(X, X, c) & = & \top \\ add(|c' \to v|, X, c) & = & c \neq c' \end{array}$$

Lemma 4. *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t, let $c \in (\mathfrak{C} \setminus t)$ that can be added to it, and w a mapping of type t. We note $w(v) = w \cup \{c \to v\}$.*

If $\forall R \in \mathcal{R}, c \notin \Gamma(R)$, then we have:

- $c \in stab(\varphi, X)$
- $\Gamma \cup \{X \to t \cup \{c\}\} \vdash \psi : t \cup \{c\}$
- $[\![\psi \bowtie |c \to v|]\!]_{V[X/\{w\}]} = [\![\psi]\!]_{V[X/\{w(v)\}]}$

Proof sketch. The first point can be proved inductively by definition of *stab* and *add*. The second point is a consequence of the first. For the third point, the only part that is not a consequence of $c \in stab(\psi, X)$ is $[\![\psi \bowtie |c \to v|]\!]_{V[X/\{w\}]} \subseteq [\![\psi]\!]_{V[X/\{w(v)\}]}$, since adding c to w could prevent mappings from being included in the result. For instance $c \in stab(\sigma_{x=c'}(X), X)$ but we do not have $[\![\sigma_{x=c'}(X) \bowtie |c \to v|]\!]_{V[X/\{w\}]} = [\![\sigma_{x=c'}(X)]\!]_{V[X/\{w(v)\}]}$ □

4.4.2 Application to rewrite rules.

Theorem 3 (Pushing joins). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t_κ and $\varphi \in \mathcal{F}[\Gamma]$ (with $X \notin free(\varphi)$) a term of type t_φ such that:*

(1) $t_\varphi \subseteq stab(\psi, X)$
(2) $\forall c \in t_\varphi \setminus t_\kappa \; add(\psi, X, c)$

Thew we have $\Gamma \vdash \mu(X = \kappa \bowtie \varphi \cup \psi) : t_\varphi \cup t_\kappa$ with for all V compatible with Γ:

$$[\![\varphi \bowtie \mu(X = \kappa \cup \psi)]\!]_V = [\![\mu(X = \kappa \bowtie \varphi \cup \psi)]\!]_V$$

Proof sketch. First we prove that $\psi \in \mathcal{F}[\Gamma \cup \{X \to t_\kappa \cup t_\varphi\}]$ by iterating Lemma 4.1. Then for each lineage $w_0, \ldots, w_n$ of $[\![\mu(X = \kappa \cup \psi)]\!]_V$ and each v compatible with w_0 we can build a lineage $(w_0 + v), \ldots, (w_n + v)$ (by iteration on Lemma 4), which proves $w + v \in [\![\mu(X = \varphi \bowtie \kappa \cup \psi)]\!]_V$.

The reverse direction is proven the same manner. □

Example 3. *If we want to compute $R_1^+(x, y) \wedge R_2(y, z)$, the naive translation would compute R_1^+ and then join with R_2. But our approach also considers the plan where we start from x, y, z such that $R_2(y, z) \wedge R_1(x, y)$ and then will discover new x by a fixpoint: $\mu\Big(X = R_1 \bowtie R_2 \cup \widetilde{\pi}_c\left(\rho_x^c(X) \bowtie \rho_y^c(R_1)\right)\Big)$*

Theorem 4 (Merging fixpoints). *Given two decomposed fixpoints $\mu(X = \kappa_1 \cup \psi_1)$ and $\mu(X = \kappa_2 \cup \psi_2)$ of types t_1 and t_2 such that:*

(1) $t_1 \cap t_2 \subseteq stab(\psi_2, X, C_2) \cap stab(\psi_1, X, C_1)$
(2) $\forall c \in t_1 \setminus t_2 \; add(\psi_2, X, c)$
(3) $\forall c \in t_2 \setminus t_1 \; add(\psi_1, X, c)$

then we have: $[\![\mu(X = \kappa_1 \cup \psi_1) \bowtie \mu(X = \kappa_2 \cup \psi_2)]\!]_V = [\![\mu(X = \kappa_1 \bowtie \kappa_2 \cup \psi_1 \cup \psi_2)]\!]_V$.

Proof sketch. The forward direction is easy: given two lineages $w_0^1, \ldots, w_n^1$ and $w_0^2, \ldots, w_m^1$ (for both $[\![\mu(X = \kappa_i \cup \psi_i)]\!]_V$) we can build a lineage $(w_0^1 + w_0^2) \ldots (w_0^1 + w_m^2) \ldots (w_n^1 + w_m^2)$.

The converse direction is more difficult but we can deinterlace the lineages and create two lineages, one for each $[\![\mu(X = \kappa_i \cup \psi_i)]\!]_V$ □

Example 4. *If we want to compute $R_1^+(x, y) \wedge R_2^+(y, z)$, the naive translation would compute both R_1^+ and R_2^+. But our approach also considers the plan where we start from x, y, z such that $R_2(y, z) \wedge R_1(x, y)$ and then will discover new x or new z by a fixpoint: $\mu(X = R_1 \bowtie R_2 \cup \psi)$ with $\psi = \widetilde{\pi}_c\left(\rho_x^c(X) \bowtie \rho_y^c(R_1)\right) \cup \widetilde{\pi}_c\left(\rho_z^c(X) \bowtie \rho_y^c(R_2)\right)$.*

Theorem 5 (Pushing antiprojections and duplications). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t_κ. Let $a, b \in \mathfrak{C}$ be such that $add(\psi, X, b) \wedge a \in stab(\psi, X)$. Then:*

$$\begin{array}{lcl} [\![\widetilde{\pi}_b(\mu(X = \kappa \cup \psi))]\!]_V & = & [\![\mu(X = \widetilde{\pi}_b(\kappa) \cup \psi)]\!]_V \\ [\![\beta_a^b(\mu(X = \kappa \cup \psi))]\!]_V & = & [\![\mu\Big(X = \beta_a^b(\kappa) \cup \psi\Big)]\!]_V \end{array}$$

Proof sketch. These properties can be proven via lineages similarly to the proofs of the other theorems. □

5 COMPARISON WITH RESPECT TO OTHER APPROACHES

Recursive queries have been well studied using various types of formalisms (see e.g. [**? ?**] for surveys). In this section, we present the main ways of evaluating such queries and why our approach can be more efficient for the specific kind of recursive queries expressed with UCRPQ. In this section, we compare first from a theoretical then an empirical point of view the several approaches.

A first line of works gathers the various methods to equip the RA with a recursive mechanism. In that regards, [4] is one of the oldest and closest to ours. They introduce a "Least Fixpoint" operator in the RA and show that selections can be pushed into these fixpoints (similar to our theorem 1) however their method produces more complex terms and handles only some type of fixpoints (for instance the recursive relation can only appear once). Authors conjecture that it is possible to push projections (similar to first item of theorem 5) but do not provide a criterion, and they do not deal with conjunction (our theorems 3 and 4). In the same line of work, authors of [**?**] provide an even more general way of pushing filters. Their algorithm works on System Graphs (SG) and computes a fixpoint of filters that is safe to recursively apply. Their works also provide an effective criterion to push projections. We believe our approach is more straightforward, furthermore their approach do not deal with join or combination of fixpoints or reversion of fixpoints.

The formalism of [**?**] that defines the *while* language seems relatively far from ours but it is actually the closest to ours. The modernization of the while$^+$ language presented in the Alice book [**?**

] is very similar to our language. Further work [**?**] actually develop on the comparison in terms of expressive power between the while and while$^+$ languages and the non-inflationary and inflationary datalog$^\neg$.

The formalism of [20] seems even closer to ours as they use a μ (inspired by the μ-calculus). However, they mainly consider the rewriting of n-ary "regular" fixpoints into queries with only transitive closures. In contrast, we use unary fixpoints with more general query forms.

5.1 Theoretical comparison

5.1.1 Automata & α-extended Relational Algebra. One way to evaluate UCRPQs is to translate individual RPQs to automata and then union-join the individual results. Another approach is to use a Relational Algebra equipped with a transitive closure operator α (thus less general than our μ operator). The automata-based method is clearly not optimal as the various RPQs are considered individually and therefore the constraints on one RPQ cannot be used on the others.

In fact, both methods can be shown to be sub-optimal on a single RPQ. Indeed, let us suppose that we are considering the regular expression $(a/b/c)^+$, the automata approach will start by computing the solution to a, then add b, then add c and recursively restart (i.e. in a computational form we have $R_{i+1} = ((R_i/a)/b)/c$). In contrast, the α-extended RA will compute once and for all $(a/b/c)$ and then compute the transitive closure (i.e. $R_{i+1} = R_i/(a/b/c)$).

The paper introducing Waveguide [26] notes that both methods force the associativity of the computation and do not test some associations such as computing $R_{i+1} = (R_i/a)/(b/c)$ which, in some cases, might be much more efficient. Their method thus mixes both methods to test more diverse plans. However they still can only consider one RPQ at a time and then join them which is sometimes suboptimal.

For instance on a query `?a :knows`$^+$ `?b, ?b :firstname :Axel`, Waveguide will not take advantage of the constraint on `?b` and will materialize the full relation `:knows`$^+$. Moreover, on a query `?a :knows`$^+$ `?b, ?b :sameAs`$^+$ `?c`, our approach will have a single fixpoint in which the number of mappings treated is exactly the number of solutions, while their approach will join the full relation `:knows`$^+$ with `:sameAs`$^+$.

5.1.2 SQL. Since its 1999 version, the SQL standard supports recursive queries via the Common Table Expressions (CTE). Note that our proposed μ-RA is not very different from SQL equipped with recursive CTE, and therefore the best plans offered by SQL 1999 are not very different from ours, however what is lacking is the possibility for SQL to rewrite such terms. Not all vendors support CTE or recursive CTE and those who do support CTE tend to consider CTE as optimization barriers. There are exceptions (such as DB2) but these vendors use a technique inspired by the Magic Set technique invented for Datalog, that we now review.

5.1.3 Datalog. Datalog is a query language supporting recursive queries. Datalog can make use of the "Magic Sets" algorithm (see [**? ?**]) to rewrite queries and limit the size of results of subqueries by pushing some type of selections and projections. The idea of the magic set is to compute, for each datalog relation, the set of "contexts" where this term will be evaluated.

For instance, in the translation of `axel:me :knows`$^+$ `?a`, the magic set method detects that, on the recursive use of `:knows`, the left side is bounded, and it will not compute the full `:knows`$^+$ relation. However, on an example corresponding to the translation of, e.g., `?a :knows`$^+$ `?b, ?b :presidentOf ?c` the magic set technique cannot be applied as all terms are free (even if we know that they are very few presidents).

5.2 A first benchmark

5.2.1 Competitors. We implemented a prototype based on the μ-RA and tested it against several other query engines:

Postgres 10.4 a popular open-source relational database. For Postgres and SQLite, the data was encoded into one table for the relation `foaf:knows` with two indexes (on its left side and its right side).

SQLite 3.24 a popular "lightweight" open-source relational database.

Openlink Virtuoso 7.2.4.2 an open-source Object-relational database management system capable of native SPARQL querying. We use the native SPARQL capabilities of Virtuoso.

ARQ 3.1.0 the query engine of Apache Jena. We use its native SPARQL capabilities to encode the query.

DLV (dec 2012) a datalog engine[2]. The translation of the SPARQL query to datalog was done manually.

Ramsdell 2.6 a datalog engine[3]. The name of the tool is simply "datalog" but for the sake of clarity we will use the name of its author: Ramsdell.

Vlog (fetched july 2018) a Column-oriented Datalog materialization for large knowledge graphs[**?**]

Not included We did not include *Neo4j* in our comparative test as its semantics is very different from the semantics of UCRPQ. We did not include Waveguide as it is not publicly available and the authors never responded.

5.2.2 Setup. For all query engines, we tried to manually reverse the direction of the fixpoint. For Virtuoso, Postgres and SQLite, it did not seem to have an effect. For ARQ, DLV, Vlog and Ramsdell the two directions had not the same behaviour and we represent these difference by using two curves for each of these engines: we therefore have ARQ_1 and ARQ_2, Ramsdell_1 and Ramsdell_2, DLV_1 and DLV_2, Vlog_1 and Vlog_2.

The graph used was a simple loop of n nodes and n edges where the transitive closure has $O(n^2)$ elements.

The translation of the query we used are depicted in figure 4.

5.2.3 Results. Figure 5 shows the running times for our benchmark query (similar to the query FOAF-1 of [6] which corresponds to our example 3).

One can experimentally observe that the other tested engines (even the datalog solvers) seem to perform, at least, quadratically in practice. Possibly at the exception of ARQ_2 which fails at 3000 nodes. This result is quite surprising since the magic set technique

[2] http://www.dlvsystem.com/dlv/
[3] http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html

SPARQL query

```
SELECT ?x ?y
{
  ?x :id 0 .
  ?x :P+ ?y .
}
```

Datalog query forward

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- path(X,Z), edge(Z,Y).
id(0).
sol :- id(Y),path(Y,_).
sol?
```

Datalog query backward

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- edge(X,Y), path(Y,Z) .
id(0).
sol :- id(Y),path(Y,_).
sol?
```

SQL query forward

```
WITH RECURSIVE path AS (
   SELECT * FROM edge
UNION ALL
  SELECT path.from AS from, edge.to AS to
  FROM path, edge
  WHERE edge.from = path.to )
SELECT NULL FROM path, id WHERE path.from = id.val
```

SQL query backward

```
WITH RECURSIVE path AS (
SELECT * FROM edge
UNION ALL
  SELECT path.to AS to, edge.from AS from
  FROM path, edge
  WHERE edge.to = path.from )
SELECT NULL FROM path, id WHERE path.from = id.val
```

Figure 4: Translations of the query used in our benchmark

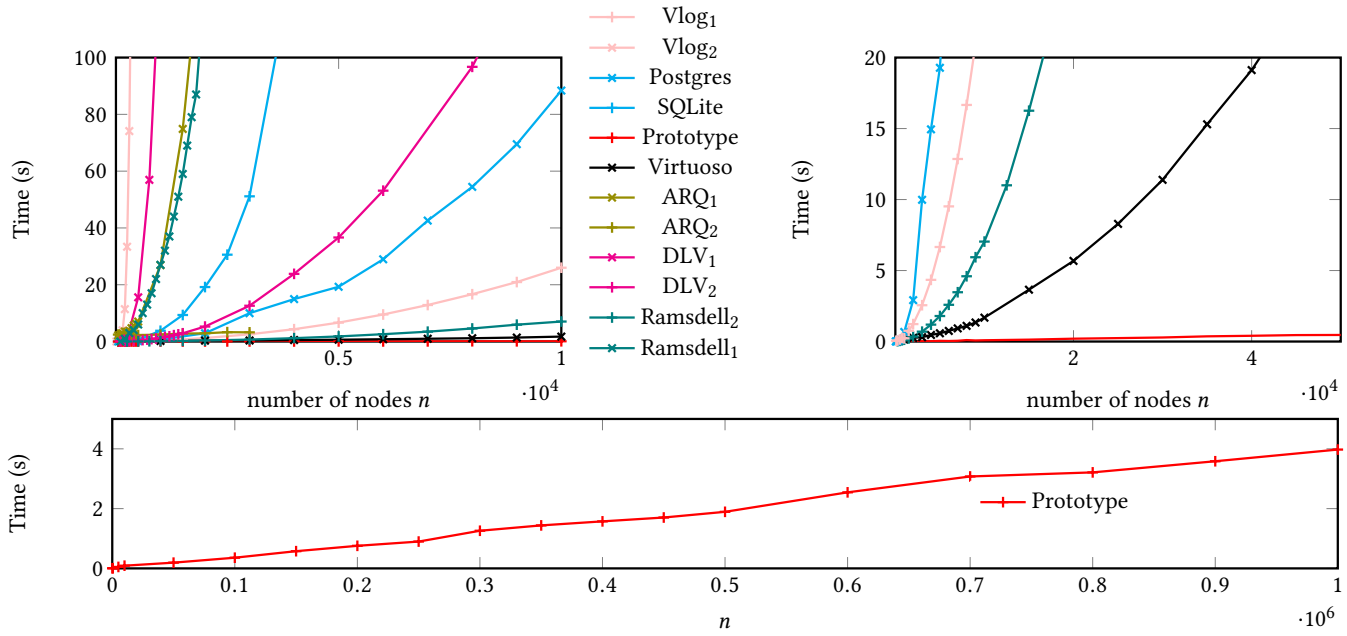

Figure 5: Evaluation time (in seconds) on a "loop" graph of size n represented with 3 windows: 10000 nodes and 100s; 50000 and 20s; 10^6 nodes and 4s .

is applied and one could expect datalog solver to be linear when the translation goes in the right way.

The more suitable plans naturally generated by our approach translate into practical performance gains: on this query, our approach takes more than a second only on the graphs that we generated when the graph has more than a million nodes.

5.3 Pushing the benchmark further

In order to get a more precise picture of how much our method improves on existing methods, we wanted to use a benchmark making use of linear recursion. The simplest use is the Union of Conjunctive Regular Path Queries. In the TASWEET paper, the benchmark is composed of a single disjunctive RPQ on a single graph, the recent gMark benchmark includes what they call "recursive" queries through a custom extension of SPARQL that allows to repeat Property Paths but only a limited number of times.

That is why we devised our own experiment to compare the tools performing the best in our first tests with our prototype based on the μ-RA. We designed our benchmark to have simple queries on large graphs.

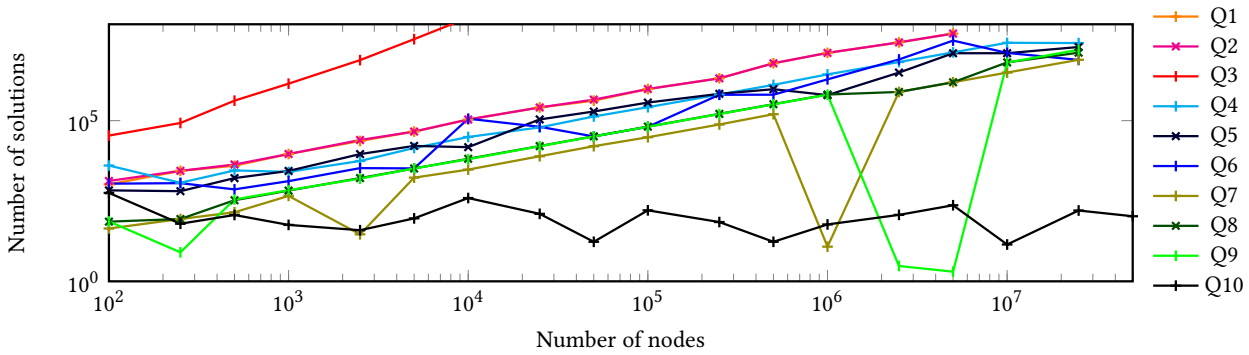

Figure 6: Number of solutions for each query and each graph size in our second benchmark.

Competitors In our second setup, we removed ARQ and SQLite, which performed very poorly in our first setup. We also removed Virtuoso that was the second top performing in our first benchmark (after our prototype), as it is limited to a very specific type of recursive queries (in our benchmark, 8 of the 10 queries were rejected by Virtuoso as they had "unbounded fixpoints").

Queries As randomly generated queries tend to have too few or too many solutions, we selected 10 queries containing between one and three fixpoints and presenting diverse behaviours on query engines. These queries are presented in Table 1, we also present the number of solutions in Figure 6 (note that Q1 is hidden by Q2). As shown, the number of solutions is roughly linear except for Q10 and Q3.

Table 1: Our queries: Pi are labels, Ni are nodes

Name	Query		
Q1	$?a$	$(P1+)/P5$	$?b$
Q2	$?a$	$(P1+)/(P5+)$	$?b$
Q3	$?a$	$(P1+)/P2$	$?b$
	$?b$	$P3+$	$?c$
Q4	$?a$	$(P4\|P5)+$	$?b$
	$?b$	$P3+$	$?c$
Q5	$?a$	$P2+$	$?b$
	$?a$	$P4+$	$?c$
	$?a$	$P5$	$N0$
Q6	$?a$	$P1 + /P2$	$?b$
	$N0$	$P3+$	$?b$
Q7	$N0$	$P1/(P2+)$	$?a$
Q8	$N0$	$(P1+)/(P2+)$	$?a$
Q9	$N0$	$P1/(P1+)$	$?a$
Q10	$?a$	$(P4+)/(P5+)/(P3+)$	$?b$

Graphs We also generated randomly graphs with varying number of nodes n (with n logarithmically spaced between $n = 100$ and $n = 50\times10^6$). In these graphs there were 5 labels $P1$ to $P5$. The edges were generated randomly such that the label Pi corresponds to $2n(1-i/5)+20$ edges. This allows for $P1+$ to be very large (roughly n^2) while $P5$ is always very selective (with 20 edges). In order to ensure that all queries had solutions, for each label Pi, we also added edges $N0 \xrightarrow{Pi} r_1$, $r_2 \xrightarrow{Pi} N0$ and $N0 \xrightarrow{Pi} N0$ (for two random nodes r_1 and r_2).

Setup For each of the graphs and each of the queries we measured the query time (it thus does not include the time to load the data). These times include the time to optimize the query. The complete results are shown in Figure 7. As for the first benchmark, we gave Postgres indexes for a fast access to both sides of edges.

Q1 and Q2 Q1 and Q2 have roughly the same solutions. This is not necessary (the queries are not equivalent) but it is due to the fact that $P1+$ captures almost all pairs of nodes while $P5+$ is very close to $P5$. Postgres has the same behaviour on both queries: by looking at the plan the Postgres optimizer selected, we see that it computes $P1+$ and $P5+$ or $P5$ separately and then joins the results. It is thus not very surprising that both queries take roughly the same time since the computation of $P5+$ is fast (as it is very small).
On the opposite, our prototype is slower on $Q2$. The query plan for $Q2$ selected by our prototype is a merged fixpoint starting from $P1/P5$ and appending/preppending $P1$ and $P5$. For $Q1$, our prototype starts from $P1/P5$ and just tries to prepend $P1$ which takes less time, hence the time difference between $Q1$ and $Q2$. However, our prototype still is the *fastest* on both queries as it does not try to materialize the full $P1+$.

Q3, Q4, Q5 On these queries, our prototype starts from a set of valid triples $?a$, $?b$, $?c$ then discovers new solutions. These plans are optimal in the sense that all partial solutions are solutions of the query but our prototype is not always the fastest.

Q3 Postgres is the *fastest* on Q3. Q3 has a lot of solutions ($\sim 10^6$ for $n = 1000$ and $\sim 10^8$ for $n = 2500$). Postgres evaluates this query by computing all the transitive closures and then joining them. The plan of Postgres is not more optimal than ours; however, Postgres is very efficient to retrieve data and join huge quantities of partial results. It thus gains on our prototype as long as the individual transitive closures are not much larger than the final result (which is what happens here and not on $Q1$ and $Q2$ with $P1+$).

Q4 On Q4, we see that many of the evaluators perform roughly the same. Despite the fact that Postgres has access to indexes

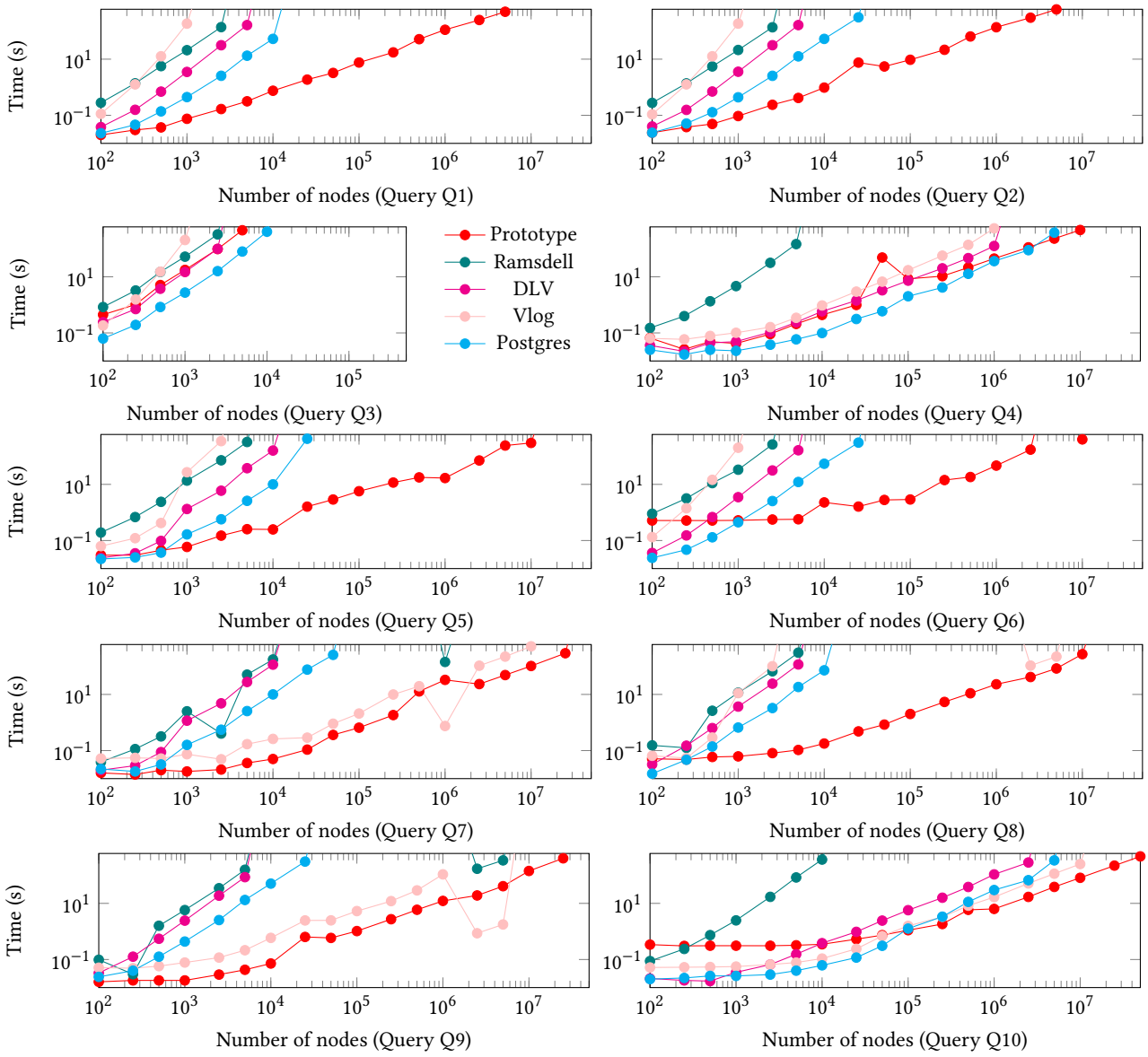

Figure 7: Query time (in seconds) as function of the size of the graph for each of our 10 queries.

and and is very efficient, our prototype is slightly faster on large graphs our prototype, this can be attributed to the optimality of our plan that starts only from valid triplets.

Q6, Q7, Q8 & Q9 These queries include a constant node ($N0$) from which our prototype will start building solutions. It is therefore not surprising that our prototype outperforms the other systems.

Q6 On Q6, all systems except our prototype seem to materialize $P1+$ which takes a lot of time (similar to $Q1$ and $Q2$).

Q7, Q8 & Q9 all these queries are queries where the datalog engines outperform Postgres. This is due to the triggering of the magic set algorithm that avoids the full materialization of the $P1+$ relation. Despite this use of the magic set, these query engines perform an order of magnitude slower than our prototype except on a few graphs where the number of solutions is very small.

Q10 This query includes three fixpoints on $P3$, $P4$ and $P5$. For each of these predicates Pi, the size of $Pi+$ stays linear in the size of Pi (contrary to e.g. $P1$ for which the size of $P1+$ grows quadratically with the graph size). This is why query engines such as Postgres perform well. Our prototype starts with pairs $(?a, ?b)$ that are solutions to $?a\ P4/P5+\ ?b$ with a fixpoint; then it computes solutions to $?a\ P4/P5 + /P3\ ?b$ (which is a valid solution) then, in a second fixpoint, grows this fixpoint by adding a $?a$ or a $?b$.

5.3.1 Synthesis. In all queries, our system performs well, beating the datalog engine on nearly all the graphs for all the queries. Furthermore, in the majority of the queries we considered, our system outperformed Postgres, often by several orders of magnitude. For the setups where our system is outperformed by Postgres, the difference is less important and can be imputed (in a large part) to the sheer performance of Postgres plan execution and not its plan space. Indeed, in Postgres, we stored triples as integers with indexes on both sides of edges while our prototype stores triples using a simple scheme and plain text files.

The easiest way to run our benchmark is to use the docker we provide: `docker run -it jachiet/mura` to start the docker and `bash run_bench.sh` to actually run the experiments. Sources to run the benchmark can be found on https://gitlab.inria.fr/tyrex-public/mu-RA and sources to our prototype can be found on https://gitlab.inria.fr/jachiet/musparql.

6 CONCLUSION

In this paper, we considered a relational algebra μ-RA equipped with a fixpoint operator that allowed us to consider new query execution plans. These plans are obtained by an extended set of rewriting rules in μ-RA.

Our model captures plans that are more efficient than the plans previously considered by existing methods. This allows querying performance to be better or at least as fast as other systems, which can be deduced theoretically and is verified experimentally.

We showed empirically that querying performance can be significantly enhanced with a prototype implementation compared to some relational engines such as Postgres, SQLite, SPARQL evaluators such as Virtuoso or Datalog Engines such as DLV.

As a future work, we plan to extend our experiments to cover more variety in graph instances and graph query workloads. Another extension we consider is to make Postgres execute directly the query plan optimized by our prototype thereby benchmarking only the optimization of the query plan (and not its execution).

REFERENCES

[1] Andrejs Abele, John P. McCrae, Paul Buitelaar, Anja Jentzsch, and Richard Cyganiak. Linking open data cloud diagram, 2017.

[2] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Shadi Ghajar-Khosravi, and Mark H. Chignell. TASWEET: Optimizing Disjunctive Path Queries in Graph Databases. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 470–473. OpenProceedings.org, 2017.

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

[4] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, New York, NY, USA, 1979. ACM.

[5] Faisal Alkhateeb and Jérôme Euzenat. Constrained regular expressions for answering rdf-path queries modulo RDFS. *IJWIS*, 10(1):24–50, 2014.

[6] Marcelo Arenas, Sebastián Conca, and Jorge Pérez. 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, pages 629–638, New York, NY, USA, 2012. ACM.

[7] Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. gMark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, 29(4):856–869, 2017.

[8] Pablo Barcelo, Diego Figueira, and Leonid Libkin. Graph logics with rational relations and the generalized intersection problem. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 115–124, Washington, DC, USA, 2012. IEEE Computer Society.

[9] Pablo Barceló, Leonid Libkin, Anthony W. Lin, and Peter T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4):31:1–31:46, December 2012.

[10] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[11] Mariano P. Consens and Alberto O. Mendelzon. Graphlog: A visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '90, pages 404–416, New York, NY, USA, 1990. ACM.

[12] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. *SIGMOD Rec.*, 16(3):323–330, December 1987.

[13] Isabel F. Cruz, Alberto O. Mendelzon, and Peter T. Wood. A graphical query language supporting recursion. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 323–330, New York, NY, USA, 1987. ACM.

[14] M. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *Proceedings 14th International Conference on Data Engineering*, pages 14–23, February 1998.

[15] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA, 2018. ACM.

[16] Georges Gardarin and Christophe de Maindreville. Evaluation of Database Recursive Logic Programs As Recurrent Function Series. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 177–186, New York, NY, USA, 1986. ACM.

[17] Andrey Gubichev, Srikanta J. Bedathur, and Stephan Seufert. Sparqling Kleene: Fast Property Paths in RDF-3x. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 14:1–14:7, New York, NY, USA, 2013. ACM.

[18] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language, W3C recommendation, March 2013.

[19] Olaf Hartig and Giuseppe Pirrò. SPARQL with property paths on the web. *Semantic Web*, 8(6):773–795, 2017.

[20] Maurice AW Houtsma and Peter MG Apers. Algebraic optimization of recursive queries. *Data & Knowledge Engineering*, 7(4):299–325, 1992.

[21] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[22] Leonid Libkin, Wim Martens, and Domagoj Vrgoč. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, March 2016.

[23] Van-Quyet Nguyen and Kyungbaek Kim. Estimating the evaluation cost of regular path queries on large graphs. In *Proceedings of the Eighth International Symposium on Information and Communication Technology*, SoICT 2017, pages 92–99, New York, NY, USA, 2017. ACM.

[24] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL Query Optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[25] Nikolay Yakovets, P Godfrey, and J Gryz. Evaluation of sparql property paths via recursive sql. 1087, 01 2013.

[26] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Waveguide: Evaluating sparql property path queries. In *EDBT*, pages 525–g528, 2015.

[27] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. Query planning for evaluating sparql property paths. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1875–1889, New York, NY, USA, 2016. ACM.

A PROOFS FOR SECTION 2 (THE μ-EXTENDED REL. ALGEBRA)

Proposition (1). *Given a database $(\mathcal{R}, \Gamma, D)$ and $\varphi \in \mathcal{F}[\Gamma]$, if $\Gamma \vdash \varphi : t$ then the relation $[\![\varphi]\!]_D$ has type t.*

Proof. Let Γ be compatible with V. The property is thus true for relation variables; it is also true for constants and by induction unions, joins, antijoins, filters, duplication or removal of columns. This leaves us with fixpoints.

Suppose $\Gamma \vdash \mu(X = \varphi) : t$. The empty set of mappings is compatible with t, thus $[\![\varphi]\!]_{V[X/\emptyset]}$ is compatible with t by induction, and thus by further induction we have $[\![\mu(X = \varphi)]\!]_V$ compatible with t. □

Proposition (2). *If $\mu(X = \varphi)$ of type t is linear, positive and non mutually recursive then the function $f(S) = [\![\varphi]\!]_{V[X/S]}$ (for S a set of mappings of type t) is such that:*

$$f(S) = f(\emptyset) \cup \bigcup_{x \in S} f(\{x\})$$

and thus f has a fixpoint with $[\![\mu(X = \varphi)]\!]_V = f^\infty(\emptyset)$.

Proof. We will first prove by induction on the size of terms the following property: given a valid term φ, for all $S \neq \emptyset$ we have $\forall m \in [\![\varphi]\!]_{V[X/S]}\ \exists w_m \in S\ m \in [\![\varphi]\!]_{V[X/\{w_m\}]}$.

- Using lemma 1 the property is clearly true for terms φ such that X is not free in φ. And the only relation variable where X appears free is X. For X the property trivially holds (with $w_m = m$).
- For unary operators $\varphi \in \{\beta_a^b(\varphi_1), \widetilde{\pi}_a(\varphi_1)$ we have $m \in [\![\varphi]\!]_{V[X/S]}$ implies the existence of $m' \in [\![\varphi_1]\!]_{V[X/S]}$ such that m is the image of m' through this operator. By the induction hypothesis, for m' there is w such that $m' \in [\![\varphi_1]\!]_{V[X/\{w_{m'}\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w_{m'}\}]}$
- For a join operator $\varphi = \varphi_1 \bowtie \varphi_2$ we have by linearity that φ is constant in X for some i (let us note $\bar{i} = 3 - i$). If $\varphi_{\bar{i}}$ is also constant in X then φ is constant in X so we can refer to the first item.
  Otherwise, $m \in [\![\varphi_{\bar{i}}]\!]_{V[X/S]}$ implies the existence of $m_1 \in [\![\varphi_1]\!]_{V[X/S]}$ and $m_2 \in [\![\varphi_2]\!]_{V[X/S]}$ such that $m = m_1 + m_2$. By induction, there exists $w_{m_{\bar{i}}}$ such that $m_{\bar{i}} \in [\![\varphi_{\bar{i}}]\!]_{V[X/\{w_{m_{\bar{i}}}\}]}$. For i we have $[\![\varphi_i]\!]_{V[X/S]} = [\![\varphi_i]\!]_{V[X/\emptyset]} = [\![\varphi_i]\!]_{V[X/\{w_{m_{\bar{i}}}\}]}$ which means that in any case $m_1 \in [\![\varphi_1]\!]_{V[X/\{w_{m_i}\}]}$, $m_2 \in [\![\varphi_2]\!]_{V[X/\{w_{m_i}\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w_{m_i}\}]}$.
- For the term $\varphi = \varphi_1 \triangleright \varphi_2$ with any mapping $m \in [\![\varphi]\!]_{V[X/S]}$ is built using at least one mapping m_1 from $[\![\varphi_1]\!]_{V[X/S]}$. By induction, we have w such that $m_1 \in [\![\varphi]\!]_{V[X/\{w\}]}$. But X does not appear free in φ_2, thus $[\![\varphi_2]\!]_{V[X/S]} = [\![\varphi_2]\!]_{V[X/\{w\}]}$ and thus $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\{w\}]}$.
- For the term $\varphi = \varphi_1 \cup \varphi_2$, $m \in [\![\varphi]\!]_{V[X/S]}$ implies $m \in [\![\varphi_1]\!]_{V[X/S]}$ or $m \in [\![\varphi_2]\!]_{V[X/S]}$. By induction we have w such that $m \in [\![\varphi_1]\!]_{V[X/\{w\}]}$ or $m \in [\![\varphi_2]\!]_{V[X/\{w\}]}$ and thus $m \in [\![\varphi]\!]_{V[X/\{w\}]}$.
- Given the term $\mu(Y = \varphi)$ we have $\mu(Y = \varphi))$ constant in X and thus the result by lemma 1.

□

Lemma (1). *Let φ be a term.*

- *If φ is recursive in X then for all V, $[\![\varphi]\!]_{V[X/\emptyset]} = \emptyset$.*
- *If φ is constant in X, then φ does not depend on X, i.e. for all S and V, $[\![\varphi]\!]_{V[X/S]} = [\![\varphi]\!]_{V[X/\emptyset]}$.*

Proof. For the constant part the result is trivial by induction.

For the recursive part, we also work by induction. It is true for base relations (constants cannot be recursive) but we need to be careful because the subterms of a recursive term can be non-recursive. By the definition of *rec* this can only happen for $\varphi_1 \bowtie \varphi_2$, but one of the φ_i has to be recursive and since the join with an empty set leads to an empty set the result holds by induction. □

Proposition (3). *A fixpoint term $\mu(X = \varphi)$, linear, positive and non mutually recursive can be rewritten to either: an empty term, a term φ with one less fixpoint or a decomposed fixpoint.*

Proof. Given a fixpoint $\mu(X = \varphi)$ we can always decompose φ into a Constant part C and a Recursive part R (possibly empty).

The idea is to prove by induction on φ that it is true for φ where X is linear, positive and non-mutually recursive:

- For a term φ constant in X the result is clear ($R = \emptyset, C = \varphi$).
- For X, $R = X, C = \emptyset$.
- For a unary operator $f(\varphi) \in \{\beta_a^b(\varphi), \widetilde{\pi}_a(\varphi), \sigma_{\mathfrak{f}}(\varphi)\}$, we have φ that can be decomposed into R_φ, C_φ the solution is $f(R_\varphi), f(C_\varphi)$ (where $f(s)$ represents $f(s)$ if s is a term and $\emptyset$ otherwise).
- For a join $\varphi_1 \bowtie \varphi_2$, let us suppose by symmetry that φ_2 is constant in X; then φ_1 can be decomposed into R, C and the result is $R \bowtie \varphi, C \bowtie \varphi$.
- For an antijoin, the same argument as joins works.
- For unions the results for subterms can be merged.
- Fixpoints are constant in X by the non mutual recursion hypothesis.

□

B DATALOG & μ-RA EXPRESSIVE POWERS

In this section we present how to translate various Datalog into μ-RA. The results presented here are not at the heart of our work and most of them are already known in the literature (with very similar statements and with similar proofs, see e.g. [**?**] or [**?**] regarding Datalog and the while$^+$ language).

The only novelty of this proof relies in the proof that the linearity of *rest-μ*-RA actually reduce the expressive power. However to understand why we need to present a translation from Datalog to μ-RA and back. We will therefore not rely on formal proofs but we will build some intuition and provide examples.

B.1 Datalog with only one IBD

We recall in this section that datalog programs can always be transformed to programs that have only one recursive rule and one output rule (this is exercise 14.17 of the alice book [**?**]).

B.1.1 Step 1: the n-aryfication. Given a Datalog program P, we can always modify P so that all rules in P are n-ary for some n. To do that we simply take n to be the maximal arity over all the rules and extend all the rules with a constant c to match this arity.

For instance:

```
Path(1,2).
Access(1).
Access(X) :- Access(Y), Path(X,Y)
```

can be made 3-ary in the following way:

```
Path(1,2,c).
Access(1,c,c).
Access(X,c,c) :- Access(Y,c,c), Path(X,Y,c)
```

B.1.2 Step 2: one rule datalog. Given a Datalog program P, we can always modify P so that there is only one recursive rule and one "output" rule in P. The idea is to first convert P into a n-ary program P' (for some n) then creates a unique $n+1$ rules that takes as its first argument the name of the rule. For instance, our running example becomes:

```
Rec(path,1,2,c).
Rec(access,1,c,c).
Rec(access,X,c,c) :- Rec(access,Y,c,c), Rec(path,X,Y,c).
Output(X) :- Rec(access,X,c,c).
```

B.2 From a derivation rule to μ-RA

It is a well-known fact that non-recurisve datalog and the relational algebra coincide (see e.g. chapter 14 of the alice book [**?**]). Given a production $head(\bar{Y}) : -body_1(\bar{X}_1), \ldots, body_k(\bar{X}_k)$ we can translate $body_1(\bar{X}_1), \ldots, body_k(\bar{X}_k)$ using $k-1$ joins between each $body_i$, renames to rename arguments of $body_i$, antiprojections to remove existential variables, and filters for constants. Finally we use joins with constants for the constants of the head and renames for the variables.

For instance, if we translate the Datalog IBD *Rec* into a term *Rec* that has 4 columns (a_1, a_2, a_3 and a_4) the translation of the body

```
Rec(access,X,c,c) :- Rec(access,Y,c,c), Rec(path,X,Y,c).
```

is $\rho^Y_{a2}\left(\widetilde{\pi}_{a_1}\left(\widetilde{\pi}_{a_3}\left(\widetilde{\pi}_{a_4}\left(\sigma_{a1=access\wedge a3=c\wedge a4=c}\left(Rec\right)\right)\right)\right)\right) \bowtie \widetilde{\pi}_{a_1}\left(\widetilde{\pi}_{a_4}\left(\rho^Y_{a3}\left(\rho^X_{a2}\left(\sigma_{a1=path\wedge a4=c}\left(Rec\right)\right)\right)\right)\right)$

The whole translation is (using *body* to denote the above term):

$\rho^{a2}_X\left(\widetilde{\pi}_Y\left(body\right)\right) \bowtie |a3 \to c| \bowtie |a4 \to c| \bowtie |a1 \to access|$

B.3 From inflationary Datalog$^\neg$ to μ-RA

Given an inflationary-Datalog$^\neg$ program P that, *w.l.o.g.*, has recursive rule *Rec* and one output rule *Output* we can translate *Rec* to a fixpoint of the form $\mu(Rec = \varphi_1 \cup \ldots \varphi_k)$ where each φ_i corresponds to one derivation of the rule *Rec*. Finally we translate each production of Output into a term ψ (where Rec is replaced by the fixpoint above) and we generate a term that is the union of all these ψ.

Given our initial example we have the term O (here we cut the translation to ease the reading):

$$
\begin{aligned}
B_1 &= |a_1 \to path| \bowtie |a_2 \to 1| \bowtie |a_3 \to 2| \bowtie |a_4 \to c| \\
B_2 &= |a_1 \to access| \bowtie |a_2 \to 1| \bowtie |a_3 \to c| \bowtie |a_4 \to c| \\
B_3 &= \rho^{a_3}_X\left(\widetilde{\pi}_Y\left(\rho^Y_{a2}\left(\widetilde{\pi}_{a_1}\left(\widetilde{\pi}_{a_3}\left(\widetilde{\pi}_{a_4}\left(\sigma_{a1=access\wedge a3=c\wedge a4=c}\left(Rec\right)\right)\right)\right)\right) \bowtie \rho^Y_{a3}\left(\rho^X_{a2}\left(\widetilde{\pi}_{a_1}\left(\widetilde{\pi}_{a_4}\left(\sigma_{a1=path\wedge a4=c}\left(Rec\right)\right)\right)\right)\right)\right)\right) \\
B_4 &= \mu(Rec = B_1 \cup B_2 \cup B3) \\
O &= \widetilde{\pi}_{a_1}\left(\widetilde{\pi}_{a_3}\left(\widetilde{\pi}_{a_4}\left(\sigma_{a_1=access}\left(B\right)_4\right)\right)\right)
\end{aligned}
$$

The semantics does coincide with inflationary-Datalog$^\neg$ because the formula $B_1 \cup B_2 \cup B_3$ captures the "immediate consequence" of the Datalog program.

B.4 From stratified Datalog to μ-RA

In a stratified Datalog program, each rule can be indexed with an integer n such that a negation of a rule indexed by k can only appear in the production rule of a term indexed with $k' > k$.

In the case of a stratified Datalog program, merging all the rules into one will break the stratification. The trick here is to operate stratum by stratum and translate the stratum i into a rule Rec_i. The resulting program will have one rule per stratum.

Just like in the inflationary case, each stratum i can be translated into a unique fixpoint $\mu(X_i = \varphi_i)$. The production rules of the stratum i can only reference to a Rec_j where $j \leq i$. We translate Rec_i into X_i and the Rec_j into $\mu(X_j = \varphi_j)$. Note that each φ_i can contain several occurrences of Rec_j with $j < i$ and that makes the translation exponential but all the fixpoints do are non mutually recursive and positive.

Let us consider the following example (already stratified):

```
Path(...) an EDB

Access_1(0).
Access_1(X) :- Access_1(Y),Path(Y,X)

Access_2(1).
Access_2(X) :- Access_2(Y),Path(Y,X), not Access_1(Y)
```

We translate Path into a term $\mu(X_0 = \varphi_0)$ (despite the fixpoint φ_0 is actually not recursive as Path is an EDB). Then we translate $Access_1$:

$$\mu\big(X_1 = |a_1 \rightarrow 0| \cup \rho_{a_2}^{a_1}\left(\widetilde{\pi}_{a_1}\left(X_1 \bowtie Path\right)\right)\big)$$

Then we translate $Access_2$ (using $Access_1$ to denote the term above) :

$$\mu\big(X_2 = |a_1 \rightarrow 1| \cup \rho_{a_2}^{a_1}\left(\widetilde{\pi}_{a_1}\left(X_2 \bowtie Path \triangleright Access_1\right)\right)\big)$$

B.5 From linear Datalog to *rest-μ*-RA

Given a linear Datalog program, we can use the stratified translation. In the resulting term each φ_i is composed of $\psi_1 \cup \ldots \psi_k$ where each of the ψ_j corresponds to a linear production rule and thus contains at most one occurrence of X_i therefore our μ-RA term is also linear (in addition to be recursive and positive as proven by the stratified translation). All in all, our term does belong to *rest-μ*-RA.

B.6 From *rest-μ*-RA to linear Datalog

This direction is actually very simple once we know how to translate a term to a Datalog program, we just need to check that the resulting term is actually linear. To translate terms into Datalog, we work bottom-up associating each subterm φ to a Datalog rule. Datalog rules have columns that are indexed (there is a first column, a second, a third, etc.) while μ-RA has column names. To handle this discrepancy, we suppose that we have calculated the type of each term (i.e. we compute a set of column names), then we order column names (any total order on the column names can be used).

The only difficulty here is the language of filters, in *rest-μ*-RA we actually impose no restriction on the filter conditions; for the translation we suppose that only the equality is used.

We thus recursively create production rules for each Datalog predicate $s_\varphi(\bar{T})$ corresponding the each term φ $s_\varphi(\bar{T})$ (where $\bar{T}$ is the ordered set of columns of the type of φ).

- For $\varphi = \varphi_1 \bowtie \varphi_2$ we create a rule for the join: $s_\varphi(\bar{T}) \leftarrow s_1(\bar{T}_1), s_2(\bar{T}_2)$.
- For $\varphi = \varphi_1 \cup \varphi_2$ we have two production rules, one for each φ_i: $s_\varphi(\bar{T}) \leftarrow s_{\varphi_i}(\bar{T}_i)$.
- For $\varphi = \varphi_1 \triangleright \varphi_2$ we create the rule $s_\varphi \leftarrow s_{\varphi_1}(\bar{T}_1), \neg s_{\varphi_2}(\bar{T}_2)$.
- For $\varphi = \sigma_{a=b}(\varphi')$ we create the rule $s_\varphi(\bar{T}_1, b, \bar{T}_2) \leftarrow s_{\varphi'}(\bar{T}_1, b, \bar{T}_2)$ if we suppose that the ordered type of φ' is $\bar{T}_1, a, \bar{T}_2$
- For $\varphi = \widetilde{\pi}_p(\varphi')$ we create the rule $s_\varphi(\bar{T}_\varphi) \leftarrow s_{\varphi'}(\bar{T}_{\varphi'})$
- For $\varphi = \beta_a^b(\varphi')$ we create the rule $s_\varphi(\bar{T}') \leftarrow s_{\varphi'}(\bar{T}_{\varphi'})$ where $\bar{T}'$ is $\bar{T}_{\varphi'}$ where we inserted a a in the place of where b will be stored.
- For $\varphi = \mu(X = \varphi')$ we create the rule $s_X(\bar{T}) \leftarrow s_{\varphi'}(\bar{T}_{\varphi'})$.
- For $\varphi = X$ we create the rule $s_X(\bar{T}) \leftarrow s_{\varphi'}(\bar{T}_{\varphi'})$.

Since the *rest-μ*-RA term is linear we can see that each production rule contain at most one subgoal that is recursive with the head.

C PROOFS FOR SECTION 4 (PROPERTIES OF THE REST-μ-RA)

Lemma (2). *Let w be a mapping and φ a term linear, positive and non mutually recursive in X. For all $m \in [\![\varphi]\!]_{V[X/\{w\}]}$ either $m \in [\![\varphi]\!]_{V[X/\emptyset]}$ or there exists $p \in d(\varphi, X)$ such that for all $c \in dom(w)$:*

$$\Big(p(c) = \bot\Big) \vee \Big(p(c) \notin dom(w)\Big) \vee \Big(m(c) = w\big(p(c)\big)\Big)$$

Proof. Let w and $m \in [\![\varphi]\!]_{V[X/\{w\}]} \setminus [\![\varphi]\!]_{V[X/\emptyset]}$. By induction:

- Since m exists, X can only be free in φ (no $|c \to v|$, no $Y \neq X$, no fixpoints).
- For a relation X, the result is clear.
- For a union we have i such that $m \in [\![\varphi_i]\!]_{V[X/\{w\}]}$ and thus the result.
- For a join $\varphi_1 \bowtie \varphi_2$ let us suppose by symmetry that φ_1 is not constant in X and that φ_2 is. We have $m_1 \in [\![\varphi]\!]_{V[X/\{w\}]}$, $d_1 \in d(\varphi_1, X) \subseteq d(\varphi_1 \bowtie \varphi_2, X)$ (with inductive hypothesis) and $m_2 \in [\![\varphi]\!]_{V[X/\emptyset]}$.
  For each $c \in dom(w)$ we either have $d_1(c) = \bot$ or $d_1(c) \notin dom(w)$ or $m_1(c) = w(d_1(c))$ and thus $m(c) = w(d_1(c))$. Note that $m_2(c)$ might or might not be defined but if $(d_1(c) \neq \bot) \wedge (p(c) \in dom(w))$ then $m_1(c)$ is also defined and $m(c) = m_1(c)$.
- For an antijoin or a filter the result is clear.
- For a duplicate or a column removal, the definition of d makes it work. Let us note $\lambda(\varphi)$ the term, we have $m \in [\![\lambda(\varphi)]\!]_{V[X/\{w\}]}$ that implies $m' \in [\![\varphi]\!]_{V[X/\{w\}]}$ and $d' \in d(\varphi, X)$ with the property. And $d' \circ \lambda$ works.

□

Lemma (3). *Given a fixpoint term $\mu(X = \varphi) \in \mathcal{F}[\Gamma]$ of type t and a mapping of type t, $w \in [\![\mu(X = \varphi)]\!]_V$ if and only if we can find a* lineage *$w_0, \dots, w_n$ for w, that is $w_0, \dots, w_n$ such that $w_0 \in [\![\varphi]\!]_{V[X/\emptyset]}$ and $w_{i+1} \in [\![\varphi]\!]_{V[X/\{w_i\}]}$.*
Furthermore for all lineages $w_0, \dots, w_n$ and all $c \in t \cap stab(\varphi, X)$, we have $w_0(c) = w(c)$.

Proof. Let $w \in [\![\mu(X = \varphi)]\!]_V$ and let n minimal such that $w \in U_n$ (as defined by the semantic). By iterating proposition 2 we $w_0, \dots, w_n = w$ as expected.
Conversely if we have such $w_0, \dots, w_n = w$ then clearly $w \in [\![\mu(X = \varphi)]\!]_V$.
Now, by Lemma 2, for each $0 \leq i \leq n-1$, the mappings w_i and w_{i+1} there is $p \in d(\varphi, X)$ such that for all $c \in stab(\varphi, X) \cap t$, $w_{i+1} = w_i(p(c) = w_i(c)$. By iteration so does w_0 and w. □

Theorem (1 Pushing filters). *Let $\mu(X = \varphi)$ be a fixpoint term, V an environment and f a filter condition with $FC(f) \subseteq stab(\varphi, X)$. Then we have $[\![\sigma_{\mathfrak{f}}(\mu(X = \varphi))]\!]_V = [\![\mu(X = \sigma_{\mathfrak{f}}(\varphi))]\!]_V$, and if $\mu(X = \varphi)$ can be decomposed into $\mu(X = \kappa \cup \psi)$, we also have $[\![\sigma_{\mathfrak{f}}(\mu(X = \kappa \cup \psi))]\!]_V = [\![\mu(X = \sigma_{\mathfrak{f}}(\kappa) \cup \psi)]\!]_V$.*

Proof. Clearly, all lineages of $[\![\mu(X = \sigma_{\mathfrak{f}}(\varphi))]\!]_V$ are lineages of $[\![\mu(X = \varphi)]\!]_V$ and all $w \in [\![\mu(X = \sigma_{\mathfrak{f}}(\varphi))]\!]_V$ pass the filter $\mathfrak{f}$.
Let $w \in [\![\sigma_{\mathfrak{f}}(\mu(X = \varphi))]\!]_V$. Let $w_0, \dots, w_n$ be a lineage of w: w passes the filter and by Lemma 3, w has the same values as all w_i on $FC(f)$; therefore $w_0, \dots, w_n$ is also a lineage of $[\![\mu(X = \sigma_{\mathfrak{f}}(\varphi))]\!]_V$. □

Theorem (2 Pushing anti-joins). *Let $\mu(X = \varphi)$ be a fixpoint term, V an environment and ψ a term of type $t \subseteq stab(\varphi, X)$ (we suppose that X is not a free variable of ψ). Then we have $[\![\mu(X = \varphi) \triangleright \psi]\!]_V = [\![\mu(X = \varphi \triangleright \psi)]\!]_V$, and if $\mu(X = \varphi)$ can be decomposed into $\mu(X = \kappa \cup \xi)$, we also have $[\![\mu(X = \kappa \cup \xi) \triangleright \psi]\!]_V = [\![\mu(X = \kappa \triangleright \psi \cup \xi)]\!]_V$.*

Proof. Clearly, all lineages of $[\![\mu(X = \varphi \triangleright \psi)]\!]_V$ are lineages of $[\![\mu(X = \varphi)]\!]_V$ and all $w \in [\![\mu(X = \varphi \triangleright)]\!]_V$ are not compatible with any mapping of $[\![\psi]\!]_V$.
Let $w \in [\![\mu(X = \varphi) \triangleright \psi]\!]_V$. Let $w_0, \dots, w_n$ be a lineage of w: w is not compatible with any mapping $w' \in [\![\psi]\!]_V$ and by Lemma 3, w has the same values as all w_i on t (the type of ψ); therefore $w_0, \dots, w_n$ is also a lineage of $[\![\mu(X = \varphi \triangleright \psi)]\!]_V$. □

Proposition (4). *Given a decomposed fixpoint of type $\{a, b\}$ of the form $\mu\big(X = \kappa \cup \widetilde{\pi}_c\left(\kappa \bowtie \rho_a^c(X)\right)\big)$, then it is equivalent to $\mu\Big(X = \kappa \cup \widetilde{\pi}_c\left(\rho_a^c\left(\rho_b^c(\kappa)\right) \bowtie \rho_b^c(X)\right)\Big)$.*

Proof. See Appendix E. □

Lemma (4). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t, let $c \in (\mathfrak{C} \setminus t)$ that can be added to it, and w a mapping of type t. We note $w(v) = w \cup \{c \to v\}$.*
If $\forall R \in \mathcal{R}, c \notin \Gamma(R)$, then we have:

- $c \in stab(\varphi, X)$
- $\Gamma \cup \{X \to t \cup \{c\}\} \vdash \psi : t \cup \{c\}$
- $[\![\psi \bowtie |c \to v|]\!]_{V[X/\{w\}]} = [\![\psi]\!]_{V[X/\{w(v)\}]}$

Proof. We will prove the result $[\![\psi]\!]_{w(v)} = [\![\psi \bowtie |c \to v|]\!]_w$ inductively on the size of ψ a term recursive in X.
Note that when a subformula ξ of ψ is constant in X we have that $[\![\xi]\!]_{V[X/\{w\}]} = [\![\xi]\!]_{V[X/\{w(v)\}]}$ by lemma 1 and we also have that c is not in the type of this ξ (since $\forall R, c \notin \Gamma R$ and the definition of *add* forbids to duplicate a column onto c). Note also that subformula that are fixpoints or constants are necessarily constant in X.
Let us explore the various cases. For the simplicity of proofs, we use $\bowtie$ and $\triangleright$ directly with sets of mappings (e.g. $A \bowtie B = \{m_A + m_B \mid m_A \in A\, m_B \in B \wedge m_A \sim m_B\}$).

- For the formula X, the result is trivial and it is the only base case (constants and other variables cannot be recursive in X).

- For $\varphi_1 \bowtie \varphi_2$, one of φ_1, φ_2 has to be constant, the other recursive. By symmetry, we suppose that φ_1 is recursive and φ_2 constant. We have $[\![\varphi_1 \bowtie \varphi_2]\!]_{V[X/\{w(v)\}]} = [\![\varphi_1]\!]_{V[X/\{w(v)\}]} \bowtie [\![\varphi_2]\!]_{V[X/\{w(v)\}]} = [\![\varphi_1 \bowtie |c \to v|]\!]_{V[X/\{w\}]} \bowtie [\![\varphi_2]\!]_{V[X/\{w\}]} = [\![(\varphi_1 \bowtie |c \to v|) \bowtie \varphi_2]\!]_{V[X/\{w\}]} = [\![(\varphi_1 \bowtie \varphi_2) \bowtie |c \to v|]\!]_{V[X/\{w\}]}$
- For $\varphi_1 \triangleright \varphi_2$ we similarly have $[\![\varphi_1 \triangleright \varphi_2]\!]_{V[X/\{w(v)\}]} = [\![\varphi_1]\!]_{V[X/\{w(v)\}]} \triangleright [\![\varphi_2]\!]_{V[X/\{w(v)\}]} = [\![\varphi_1 \bowtie |c \to v|]\!]_{V[X/\{w\}]} \triangleright [\![\varphi_2]\!]_{V[X/\{w\}]} = [\![(\varphi_1 \bowtie |c \to v|) \triangleright \varphi_2]\!]_{V[X/\{w\}]} = [\![(\varphi_1 \triangleright \varphi_2) \bowtie |c \to v|]\!]_{V[X/\{w\}]}$ and the last line that uses the commutativity of $\triangleright$ over $\bowtie$ is only true because c cannot be in the type of φ_2.
- For $\sigma_{\mathfrak{f}}(\varphi')$, $\widetilde{\pi}_a(\varphi')$ and $\beta_a^b(\varphi)$ the result comes easily as $c \notin \{a, b\} \cup FC(f)$.

□

Theorem (3 Pushing joins). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t_κ and $\varphi \in \mathcal{F}[\Gamma]$ (with $X \notin free(\varphi)$) a term of type t_φ such that:*

(1) $t_\varphi \subseteq stab(\psi, X)$
(2) $\forall c \in t_\varphi \setminus t_\kappa$ $add(\psi, X, c)$

Thew we have $\Gamma \vdash \mu(X = \kappa \bowtie \varphi \cup \psi) : t_\varphi \cup t_\kappa$ with for all V compatible with Γ:

$$[\![\varphi \bowtie \mu(X = \kappa \cup \psi)]\!]_V = [\![\mu(X = \kappa \bowtie \varphi \cup \psi)]\!]_V$$

Proof. Lemma 4 ensures us that $\Gamma \cup \{X \to t_\varphi \cup t_\kappa\} \vdash \psi : t_\varphi \cup t_\kappa$, and thus $\Gamma \vdash \mu(X = \varphi \bowtie \kappa \cup \psi) : t_\varphi \cup t_\kappa$.

Then if we take a lineage $w_0 \dots w_n$ of $[\![\mu(X = \kappa \cup \psi)]\!]_V$ and there exists $u \in [\![\varphi]\!]_V$ compatible with w_n then $t_\varphi \subseteq stab(\psi, X)$ ensures us that u is compatible with all w_i.

Then by iterating Lemma 4, for each i and for each $c \in t_\varphi \setminus t_\kappa$, we have that $w_0(u), \dots, w_n(u)$ is a valid lineage of $[\![\mu(X = \kappa \bowtie \varphi \cup \psi)]\!]_V$ and reciprocally. □

Theorem (4 Merging fixpoints). *Given two decomposed fixpoints $\mu(X = \kappa_1 \cup \psi_1)$ and $\mu(X = \kappa_2 \cup \psi_2)$ of types t_1 and t_2 such that:*

(1) $t_1 \cap t_2 \subseteq stab(\psi_2, X, C_2) \cap stab(\psi_1, X, C_1)$
(2) $\forall c \in t_1 \setminus t_2$ $add(\psi_2, X, c)$
(3) $\forall c \in t_2 \setminus t_1$ $add(\psi_1, X, c)$

then we have: $[\![\mu(X = \kappa_1 \cup \psi_1) \bowtie \mu(X = \kappa_2 \cup \psi_2)]\!]_V = [\![\mu(X = \kappa_1 \bowtie \kappa_2 \cup \psi_1 \cup \psi_2)]\!]_V$.

Proof. For $i \in \{1, 2\}$, let $w_0, \dots, w_{n_i}$ be a lineage of $[\![\mu(X = \kappa^i \cup \psi^i)]\!]_V$ with w_{n_1} compatible with w_{n_2}; we can easily construct a lineage of size $n_1 + n_2$ of the form $(w_0^1 + w_0^2) \dots (w_{n_1}^1 + w_0^2) \dots (w_{n_1}^1 + w_{n_2}^2)$ and for the same reason as the last theorem, it holds.

Now let us take a lineage $w_0, \dots, w_n$ of $[\![\mu(X = \kappa_1 \bowtie \kappa_2 \cup \psi_1 \cup \psi_2)]\!]_V$. We decompose w_i into $w_i^1 + w_i^2$ where w_i^j is the restriction of w_i to the type of κ_j. Those w_i^j are not necessarily forming a lineage but we consider the subsequence containing w_0^i and for each $i > 0$ w_j^i when $w_i^j \in [\![\psi_i]\!]_{V[X/\{w_{i-1}^j\}]}$. Then by the theorem condition when $w_i^j \in [\![\psi_i]\!]_{V[X/\{w_{i-1}^j\}]}$ we have $w_i^{\bar{j}} = w_{i-1}^{\bar{j}}$. The two resulting sequences are thus lineages and we have the expected theorem.

□

Theorem (5 Pushing antiprojections and duplications). *Let $\mu(X = \kappa \cup \psi) \in \mathcal{F}[\Gamma]$ be a decomposed fixpoint of type t_κ. Let $a, b \in \mathfrak{C}$ be such that $add(\psi, X, b) \wedge a \in stab(\psi, X)$. Then:*

$$\begin{aligned} [\![\widetilde{\pi}_b(\mu(X = \kappa \cup \psi))]\!]_V &= [\![\mu(X = \widetilde{\pi}_b(\kappa) \cup \psi)]\!]_V \\ [\![\beta_a^b(\mu(X = \kappa \cup \psi))]\!]_V &= [\![\mu\Big(X = \beta_a^b(\kappa) \cup \psi\Big)]\!]_V \end{aligned}$$

Proof.
- This is a conclusion of lemma 4. Let $w_0, \dots, w_n$ be a lineage of $[\![\mu(X = \widetilde{\pi}_c(\kappa) \cup \psi)]\!]_V$ there exists v such that $w_0(v) \in [\![\kappa]\!]_{V[X/\emptyset]}$ and if we have $w_i(v)$ we can find $w_{i+1}(v) \in [\![\psi]\!]_{V[X/\{w_i(v)\}]}$ by lemma 4. In the end we have a lineage for $w(v) \in [\![\mu(X = \kappa \cup \psi)]\!]_V$ and which means $w \in \widetilde{\pi}_c(\mu(X = \kappa \cup \psi))$.
  Notice that lemma 4 gives an equality therefore this is a bijection between lineage and also proves the converse way.
- Let $w \in [\![\beta_a^b(\mu(X = \kappa \cup \psi))]\!]_V$ we have $w_0, \dots, w_n$ a lineage of $[\![\mu(X = \kappa \cup \psi)]\!]_V$ such that $w = w_n$ except $w(b) = w(a)$. Let $w_i' = w_i + \{b \to w_0(a)\}$ we have that $w_0', \dots, w_n'$ is a lineage of $[\![\mu\Big(X = \beta_a^b(\kappa) \cup \psi\Big)]\!]_V$ with lemma 4 but $w_i'(a) = w_0'(a) = w_n'(a) = w_n'(b)$ therefore $w = w_n' \in [\![\mu\Big(X = \beta_a^b(\kappa) \cup \psi\Big)]\!]_V$.
  Conversely, let $w \in [\![\mu\Big(X = \beta_a^b(\kappa) \cup \psi\Big)]\!]_V$ we have $w_0, \dots, w_n$ a lineage of $[\![\mu\Big(X = \beta_a^b(\kappa) \cup \psi\Big)]\!]_V$ such that $w = w_n$. Let $w_i' = w_i$ but where we removed the column b we have that $w_0', \dots, w_n'$ is a lineage of $[\![\mu(X = \kappa \cup \psi)]\!]_V$ with lemma 4 and that $w_{i+1}'(b) = w_i'(b) = w_0'(b) = w_n'(b)$ therefore $w \in [\![\beta_a^b(\mu(X = \kappa \cup \psi))]\!]_V$.

- The argument is the same as the one of theorem 3: let $w_0, \ldots, w_n$ be a lineage of $[\![\mu(X = \kappa \cup \psi)]\!]_V$, then $u \in [\![\varphi]\!]_V$ is compatible with w_n if and only if w_0 is which $w_n \in [\![\mu(X = \kappa \cup \psi) \triangleright \varphi]\!]_V$ if and only if $w_0 \in [\![\kappa \triangleright \varphi]\!]_V$ and thus if and only if $w_n \in [\![\mu(X = \kappa \triangleright \varphi \cup \psi)]\!]_V$. □

D DETAILS OF OUR SECOND BENCHMARK

Table 2: Our ten queries: Pi are labels, Ni are nodes

Name	Query		
Q1	$?a$	$(P1+)/P5$	$?b$
Q2	$?a$	$(P1+)/(P5+)$	$?b$
Q3	$?a$	$(P1+)/P2$	$?b$
	$?b$	$P3+$	$?c$
Q4	$?a$	$(P4\|P5)+$	$?b$
	$?b$	$P3+$	$?c$
Q5	$?a$	$P2+$	$?b$
	$?a$	$P4+$	$?c$
	$?a$	$P5$	$N0$
Q6	$?a$	$P1 + /P2$	$?b$
	$N0$	$P3+$	$?b$
Q7	$N0$	$P1/(P2+)$	$?a$
Q8	$N0$	$(P1+)/(P2+)$	$?a$
Q9	$N0$	$P1/(P1+)$	$?a$
Q10	$?a$	$(P4+)/(P5+)/(P3+)$	$?b$

Table 3: Number of solutions for each query and each graph

Query	Number of solutions for n =																	
	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
Q1	1035	2742	3960	9160	23307	45785	108597	257829	419302	964676	2077707	6104086	12851317	27285381	51323155	59953009	39624142	?
Q2	1330	2742	4316	9160	24969	45786	108597	257829	451550	964678	2077707	6104086	12851317	27285381	51323155	46784404	22993679	?
Q3	34307	84611	419140	1415274	7656802	34385488	160647218	?	?	?	?	?	?	?	?	?	?	?
Q4	4015	1167	2826	2544	5563	14106	31012	61695	134770	262135	650429	1297915	2753082	6692132	13334968	26476752	26006107	?
Q5	677	639	1631	2722	9114	16364	14987	109289	192653	365625	683670	952857	624038	3134311	12531942	12560576	19556646	?
Q6	1093	1136	727	1311	3334	3269	114987	64457	32264	64347	639292	642534	1927697	8025112	31038127	12835432	7774933	?
Q7	44	87	141	455	29	1686	3001	7805	16052	30445	75998	158807	12	783574	1566480	3140131	7839973	?
Q8	72	87	334	666	1634	3252	6504	16190	32622	65052	162602	325348	650274	783574	1566480	6499291	13054450	?
Q9	73	8	353	685	1569	3258	6320	16031	32154	64723	160677	321923	643960	3	2	6417508	16042045	?
Q10	565	62	116	57	39	91	391	126	17	162	70	17	59	117	233	14	160	106

GRADES-NDA'18, June 10–15, 2018, Houston, TX, USA

Table 4: Time in milliseconds to evaluate queries in each query engine, T means timeout (thus $>$ 600000 ms = 10 min)

Our prototype																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	20	30	37	75	167	315	750	1869	3231	7589	17468	51508	111490	248563	488896	T	T	T
$Q2$	24	38	49	94	237	415	970	7546	5488	9510	21556	64344	137279	300510	593119	T	T	T
$Q3$	445	1026	5072	17570	97371	454282	T	T	T	T	T	T	T	T	T	T	T	T
$Q4$	66	26	49	42	91	209	437	989	48561	8435	10527	21361	44554	111947	229056	464465	T	T
$Q5$	30	30	45	59	149	253	248	1609	2886	5702	11601	17736	16789	70998	241506	299637	T	T
$Q6$	516	515	515	522	559	570	2275	1618	2786	2892	14230	18266	46627	172920	T	400496	T	T
$Q7$	16	14	20	18	21	36	50	108	360	653	1835	12773	32955	23330	48907	102896	293638	T
$Q8$	51	48	59	62	81	105	178	482	841	2008	5395	10991	22993	41633	84018	272161	T	T
$Q9$	16	18	18	18	29	43	72	636	587	1029	2726	5920	12232	19133	40918	142755	402768	T
$Q10$	339	305	310	310	309	321	349	530	737	1106	1836	5878	6344	16989	38571	81378	221415	471908

Ramsdell datalog																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	280	1376	5649	20907	138423	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q2$	277	1369	5503	21234	136629	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q3$	836	3250	14843	52035	323688	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q4$	150	397	1339	4655	31442	146118	T	T	T	T	T	T	T	T	T	T	T	T
$Q5$	192	684	2389	13600	72106	323422	T	T	T	T	T	T	T	T	T	T	T	T
$Q6$	901	3163	11094	33286	264580	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q7$	39	113	319	2503	403	50738	175707	T	T	T	T	T	143564	T	T	T	T	T
$Q8$	153	125	2619	11550	66633	307356	T	T	T	T	T	T	T	T	T	T	T	T
$Q9$	98	30	1593	5770	34679	157606	T	T	T	T	T	T	T	171043	340492	T	T	T
$Q10$	88	238	739	2474	17056	83787	363832	T	T	T	T	T	T	T	T	T	T	T

Postgres																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	23	46	138	445	2557	13266	52817	T	T	T	T	T	T	T	T	T	T	T
$Q2$	24	51	129	438	2551	12579	52743	309599	T	T	T	T	T	T	T	T	T	T
$Q3$	63	193	837	2731	15898	79327	403633	T	T	T	T	T	T	T	T	T	T	T
$Q4$	25	17	25	23	38	59	99	313	592	2021	4147	12763	36110	88753	373743	T	T	T
$Q5$	22	25	37	165	570	2589	10031	422135	T	T	T	T	T	T	T	T	T	T
$Q6$	24	47	130	446	2547	12152	54281	307631	T	T	T	T	T	T	T	T	T	T
$Q7$	22	18	32	159	553	2549	10008	76641	256535	T	T	T	T	T	T	T	T	T
$Q8$	15	47	139	661	3264	18359	72266	T	T	T	T	T	T	T	T	T	T	T
$Q9$	24	40	126	436	2541	13211	50576	308115	T	T	T	T	T	T	T	T	T	T
$Q10$	20	22	26	26	29	40	62	117	308	1264	3353	11291	29864	67052	340287	T	T	T

DLV																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	38	157	698	3506	31378	163923	T	T	T	T	T	T	T	T	T	T	T	T
$Q2$	40	156	704	3568	31387	163717	T	T	T	T	T	T	T	T	T	T	T	T
$Q3$	242	707	3744	14769	98659	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q4$	36	22	44	49	107	239	586	1425	3294	7280	20020	46237	125058	T	T	T	T	T
$Q5$	24	35	97	1325	5949	37692	160384	T	T	T	T	T	T	T	T	T	T	T
$Q6$	36	152	684	3499	31347	163998	T	T	T	T	T	T	T	T	T	T	T	T
$Q7$	20	29	88	1161	4811	27714	114981	T	T	T	T	T	T	T	T	T	T	T
$Q8$	32	149	632	3682	24174	116852	T	T	T	T	T	T	T	T	T	T	T	T
$Q9$	33	126	548	2427	18793	85968	T	T	T	T	T	T	T	T	T	T	T	T
$Q10$	21	18	17	35	66	154	382	966	2431	5714	15888	38375	107760	285948	T	T	T	T

Vlog																		
n	100	250	500	1000	2500	5000	10000	25000	50000	100000	250000	500000	1000000	2500000	5000000	10000000	25000000	50000000
$Q1$	113	1265	12685	184598	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q2$	108	1265	12664	184995	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q3$	184	1559	15168	204207	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q4$	64	59	78	101	161	345	950	2956	6689	17157	56588	137383	531602	T	T	T	T	T
$Q5$	63	121	420	27047	347451	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q6$	132	1410	14498	201749	T	T	T	T	T	T	T	T	T	T	T	T	T	T
$Q7$	52	55	52	74	49	169	260	288	904	2084	9917	19587	749	105329	222389	516499	T	T
$Q8$	65	54	290	11139	100941	T	T	T	T	T	T	T	T	106399	221938	T	T	T
$Q9$	51	49	58	78	117	214	589	2465	2479	5311	12032	29011	107995	862	1770	T	T	T
$Q10$	52	53	54	56	66	78	108	239	687	1577	3293	8259	16715	51337	114233	252075	T	T

E REVERSING FIXPOINTS

E.1 Motivation

As we have seen, pushing filters, selections and joins into a fixpoint $\mu(X = \varphi)$ depends on φ and the derivations of φ.

E.2 Simple case

Let us consider a fixpoint of the form $\mu\left(X = \varphi \cup \widetilde{\pi}_c\left(\rho_a^c(\varphi) \bowtie \rho_b^c(X)\right)\right)$ of type $\{a, b\}$ and that a, b and c are all distinct. φ represents a binary relation from a to b and the fixpoint computes its transitive closure. Therefore the role of a and b are symmetrical and this fixpoint computes the same relation as $\mu\left(X = \varphi \cup \widetilde{\pi}_c\left(\rho_b^c(\varphi) \bowtie \rho_a^c(X)\right)\right)$.

E.3 Simple case extended to n columns

Let us now consider a fixpoint of the form $\mu\left(X = \varphi \cup \widetilde{\pi}_{c_1}\left(\dots \widetilde{\pi}_{c_n}\left(\rho_{a_1}^{c_1}\left(\dots \rho_{a_n}^{c_n}(\varphi)\right) \bowtie \rho_{b_1}^{c_1}\left(\dots \rho_{b_n}^{c_n}(X)\right)\right)\right)\right)$ and let us suppose that the type is $\{a_1, \dots, a_n, b_1, \dots, b_n\}$ with the (a_i), (b_i) and (c_1) all distinct. φ represents a binary relation from the n-uplet $a_1, \dots, a_n$ to the n-uplet $b_1, \dots, b_n$ and the fixpoint computes its transitive closure. Similarly as in the simple case the role of (a_i) and (b_i) are symmetrical and this fixpoint computes the same set as
$\mu\left(X = \varphi \cup \widetilde{\pi}_{c_1}\left(\dots \widetilde{\pi}_{c_n}\left(\rho_{b_1}^{c_1}\left(\dots \rho_{b_n}^{c_n}(\varphi)\right) \bowtie \rho_{a_1}^{c_1}\left(\dots \rho_{a_n}^{c_n}(X)\right)\right)\right)\right)$.

E.4 Via an unfolding of a fixpoint over one column

Given a fixpoint of the form $\mu\left(X = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(X) \bowtie \varphi\right)\right)$ with type $\{a, b\}$, and type of Γ is $\{a, d\}$ and a, b, d all different.

This fixpoint actually computes the reflexive transitive closure of the relation between a and b induced by φ and then joins the relation between a and b induced by ψ which gives an overall relation between a and d.

From a relation point of view, we can replace the reflexive transitive closure with the union of the transitive closure plus the identity relation. With a fresh c (i.e. $c \notin \{a, b, d\}$) we have:

$\mu\left(X = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(X) \bowtie \varphi\right)\right) = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(\psi) \bowtie \mu\left(X = \varphi \cup \widetilde{\pi}_c\left(\rho_a^c(X) \bowtie \rho_b^c(\varphi)\right)\right)\right)$

or using the reversion presented earlier:

$\mu\left(X = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(X) \bowtie \varphi\right)\right) = \psi \cup \widetilde{\pi}_b\left(\rho_a^b(\psi) \bowtie \mu\left(X = \varphi \cup \widetilde{\pi}_c\left(\rho_b^c(X) \bowtie \rho_a^c(\varphi)\right)\right)\right)$

E.5 Via the unfolding of a fixpoint over several columns

Just as before but a is $a_1, \dots, a_n$, b is $b_1, \dots, b_n$ and d is $d_1, \dots, d_k$ (note that we do not need to have $k = n$). Given a fixpoint of the form $\mu\left(X = \psi \cup \widetilde{\pi}_{b_1}\left(\dots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\dots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\right)$ with $P(\varphi, \Gamma) = C(\varphi, \Gamma) = \{a_1, \dots, c_n, b_1, \dots, b_n\}$, $P(\psi, \Gamma) = C(\psi, \Gamma) = \{a_1, \dots, a_n, d_1, \dots, d_k\}$ and a_i, b_j, d_l all different.

We have $\mu\left(X = \psi \cup \widetilde{\pi}_{b_1}\left(\dots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\dots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\right) =$

$$\psi \cup \widetilde{\pi}_{b_1}\left(\dots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\dots \rho_{a_n}^{b_n}(\psi)\right) \bowtie \mu\left(X = \varphi \cup \widetilde{\pi}_{c_1}\left(\dots \widetilde{\pi}_{c_n}\left(\rho_{a_1}^{c_1}\left(\dots \rho_{a_n}^{c_n}(X)\right) \bowtie \rho_{b_1}^{c_1}\left(\dots \rho_{b_n}^{c_n}(\varphi)\right)\right)\right)\right)\right)\right)$$

or using the reversion presented earlier:

$\mu\left(X = \psi \cup \widetilde{\pi}_{b_1}\left(\dots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\dots \rho_{a_n}^{b_n}(X)\right) \bowtie \varphi\right)\right)\right) =$

$$\psi \cup \widetilde{\pi}_{b_1}\left(\dots \widetilde{\pi}_{b_n}\left(\rho_{a_1}^{b_1}\left(\dots \rho_{a_n}^{b_n}(\psi)\right) \bowtie \mu\left(X = \varphi \cup \widetilde{\pi}_{c_1}\left(\dots \widetilde{\pi}_{c_n}\left(\rho_{b_1}^{c_1}\left(\dots \rho_{b_n}^{c_n}(X)\right) \bowtie \rho_{a_1}^{c_1}\left(\dots \rho_{a_n}^{c_n}(\varphi)\right)\right)\right)\right)\right)\right)$$