Efficient Enumeration of Recursive Plans in Transformation-based Query Optimizers

Amela Fejza, Pierre Genevès, Nabil Layaïda

Efficient Enumeration of Recursive Plans in Transformation-based Query Optimizers

Amela Fejza*, Pierre Genevès†, Nabil Layaïda‡
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
Email: *amela.fejza@inria.fr, †pierre.geneves@inria.fr, ‡nabil.layaida@inria.fr

***Abstract*—Query optimizers built on the transformation-based Volcano/Cascades framework are used in many database systems. Transformations proposed earlier on the logical query dag (LQDAG) data structure, which is key in such a framework, focus only on recursion-free queries. In this paper, we propose the recursive logical query dag (RLQDAG) which extends the LQDAG with the ability to capture and transform recursive queries, leveraging recent developments in recursive relational algebra. Specifically, this extension includes: (i) the ability of capturing and transforming sets of recursive relational terms thanks to (ii) annotated equivalence nodes used for guiding transformations that are more complex in the presence of recursion; and (iii) RLQDAG rewrite rules that transform sets of subterms in a grouped manner, instead of transforming individual terms in a sequential manner; and that (iv) incrementally update the necessary annotations. Core concepts of the RLQDAG are formalized using a syntax and formal semantics with a particular focus on subterm sharing and recursion. The result is a clean generalization of the LQDAG transformation-based approach, enabling more efficient explorations of plan spaces for recursive queries. An implementation of the proposed approach shows significant performance gains compared to the state-of-the-art.**

I. Introduction

Recursive queries enable powerful information extraction, especially from linked data structures such as trees and graphs. However, important data management system components, such as the widely used Volcano framework [27], were designed for recursion-free queries. A typical transformation-based query optimizer operates by (i) translating a query into a relational algebraic term, (ii) applying algebraic transformations in order to search for equivalent yet more efficient evaluation plans, during a so-called *plan enumeration phase*, (iii) executing the query by running one of the explored plans.

Works on extending relational algebra (RA) with recursion [2], [10] resulted in powerful recursive relational algebras [4], [5], [32], capable of capturing queries with transitive closures [4] and even more general forms of recursion [5], [32]. This line of works recently led to μ-RA [32] which provides a rich set of rewrite rules for recursive terms enabling efficient recursive evaluation plans not available with earlier approaches.

The enumeration phase is crucial as it may produce terms which are drastically more efficient. It has been heavily researched for recursion-free queries. It is common practice to allocate a time budget for this enumeration phase, as it is notoriously known that exhaustive plan space explorations may not be feasible in practice for certain queries. The faster we generate the space of equivalent plans, the more likely we will be able to find terms with more efficient evaluation.

With recursion, plan spaces are often significantly larger than in the non-recursive setting, due to new interplays between recursive and non-recursive operators. The efficiency of recursive plan enumeration becomes critical. The speed of plan enumeration directly determines whether query evaluation plans enabled by, e.g. μ-RA [32], are within range or theoretically reachable, but still out of reach for a practical query optimizer. This motivates the search for efficient methods for enumerating recursive plans.

Contributions: We present the recursive logical query dag (RLQDAG), which extends Volcano's LQDAG [27] and the μ-RA framework [32] for the purpose of efficiently enumerating recursive query plans. Contributions include the first extension of the LQDAG with the support of recursive terms; a formalization of important RLQDAG concepts in terms of formal syntax and semantics, with a particular focus on the sharing of common subterms in the presence of recursion; RLQDAG transformations that generalize rewrite rules for individual recursive terms of [32] and enable efficient and grouped transformations of sets of recursive terms. The RLQDAG relies on a concept of annotated equivalence nodes with incremental updates, used for guiding transformations of recursive subterms. Contributions also include a complete implementation of the proposed approach; and experimental assessments using third-party queries on synthetic and real datasets.

II. Background and Related Work

A. Approaches in recursive query optimization

Recursion is considered in only a small fraction of the numerous works on query optimization. Three main lines of work with recursive queries can be identified:

- The Datalog line of works [7], [16], [20], [31], [45], [49], [55], [57] developed methods for optimizing recursive queries formulated in Datalog: magic-sets [9], [23], [42], demand transformations [52], automated reversals [40], and the FGH rule [59]. Although the syntax of Datalog greatly differs from RA, the effects of magic-sets [9], [23], [42] and of demand transformations [52]

This research has been partially supported by the ANR project GraphRec (ANR-23-CE23-0010).

are comparable to pushing certain kinds of selections and projections. These techniques are very sensitive to whether the Datalog program is written in a left-linear or right-linear manner, but one can use the automated reversal technique proposed in [40] to fully exploit them. The optimization framework proposed in [59] gathers magic-sets, semi-naive evaluation and proposes a new FGH rule for optimizing recursive Datalog programs with aggregations. Datalog engines do not explore plan spaces but use heuristics to find a good plan to evaluate queries. However, currently, no matter which combination of existing Datalog optimizations a Datalog engine implements, it will not be able to find plans where recursions have been merged automatically similar to those found by the μ-RA approach [32]. This is because, currently, in a Datalog program corresponding to the optimized translation of A^+/B^+ at least one of the two transitive closures A^+ or B^+ will be fully materialized (even if there is no solution to A^+/B^+). On real datasets, this can make Datalog query evaluation an order of magnitude slower than query evaluation with RA-based systems, as noticed in [32].

- The line of works based on relational algebra [4], [5], [14], [32], [33] attempts to extend relational algebra with operators to capture forms of recursion. For instance, α-RA [4] extends RA with an operator to capture transitive closures. LFP-RA [5] proposes a more general least fixpoint operator, and an extension of this work gave effective criteria for optimization in its presence [33]. Recently, μ-RA [32] proposed to extend RA with a μ operator which is also a fixpoint with an appropriate set of restrictions. This enables μ-RA to combine all earlier RA-based recursion optimization rules in the same framework [32], while adding new rewrite rules for recursive terms, in particular for merging recursions. This makes μ-RA the most advanced system for RA-based recursive query optimization as it can generate plans not reachable by other approaches. The approach that we propose in this paper extends μ-RA with a new method for enumerating recursive plans much more efficiently, thanks to a generalization of the rules presented in μ-RA so that the generalized rules (presented in Section IV) apply directly on a factorized representation on the recursive plan space. Among the benefits, this makes it possible to apply transformations on sets of algebraic terms at once (instead of successively on individual terms), and to exploit the sharing of common subterms to avoid redundant computations.
- Ad-hoc optimizations for regular queries. Both Datalog-based and RA-based approaches can capture queries with expressive forms of recursion, going beyond regularity. Several works have focused on optimizing queries in which recursion is restricted to regular patterns, such as regular path queries (RPQs). Automata based approaches have been developed to answer RPQ queries [24], [34], [64]. A hybrid approach that combines finite state machines and α-RA is presented in [63] and extended in [3]. All these works are limited to RPQs and their unions or conjunctions. In comparison, the work we present in this paper supports more expressive forms of recursion, that may include non-regular patterns (see e.g. the experimental section V with queries of the form $A^n B^n$ for instance).

B. Limits of the RA-based approach

The RA line of work offers several advantages including high expressivity and rich plan spaces. In addition, it can be seen as an interesting approach for extending the (non-recursive) RA approach already widely used in RDBMS implementations. One main limit however, is that the whole approach critically depends on the ability to quickly enumerate plans during the plan exploration phase. While plan enumeration has not been studied yet in the presence of recursion, it has been extensively studied for recursion-free queries.

C. Plan enumeration for non-recursive (SPJ) queries

Most of the works on plan enumeration focus on select-project-join (SPJ) queries. Techniques proposed for SPJ can be divided into two main groups: bottom-up and top-down approaches. Bottom-up approaches generate the plan space by starting from the leaves (initial relations) and going up in the tree of operators when progressively exploring alternate combinations of operators. In contrast, top-down approaches start from the root and recursively explore subbranches in search for possible alternatives.

1) Bottom-up approaches: Pioneering and seminal works on the bottom-up approach were proposed in [44] with refinements in execution strategies [38] and the Starburst plan generator [29], [37]. More recently, E-Graphs [61] and equality saturation techniques [51] introduced in the context of SMT solvers have been used in the database community to optimize recursion-free relational and tensor algebra [43], [58]. E-graph [61] basically corresponds to the LQDAG [27] (also known as AND-OR-DAG), as noticed in [58]. In comparison to LQDAG and E-Graph, that do not support recursive queries, RLQDAG adds the support for recursion. E-Graph is fundamentally bottom-up, whereas RLQDAG relies on top-down search to prune the search space. The top-down aspect of RLQDAG is essential to solve the problem of exploring recursive plans, since transformations of sets of recursive subterms critically rely on a complete view of recursive patterns.

2) Top-down approaches: The top-down approach was introduced with EXODUS [26] and the seminal works on Volcano [27] and Cascades [25] that added memoization to increase efficiency. The purpose of memoization is to reuse information already found in order to avoid redundant calculations. An advantage of the top-down approach is that it makes it possible to use branch and bound pruning, as noticed in [47].

All the previous approaches focus mainly on SPJ queries, with some extensions to support outer joins [17], [18], [21], [22], [39] and aggregations [13].

Very few works consider other operators, as noticed in [12] and [46]. To the best of our knowledge, plan enumeration for transformation-based query optimizers has not been studied yet in the presence of recursive operators.

Union and recursion greatly extend the expressive power of SPJ queries. However, they not only make plan enumeration significantly more complex, but they also generate significantly larger plan spaces. This is because their addition generates many new possible combinations to be explored due to new interplays, for instance between unions and joins (e.g. distributivity of natural join over union) or between recursions and joins. This worsens the combinatorics of plans to be enumerated and motivates even more the interest of finding efficient techniques for enumerating recursive plans.

D. The logical query dag (LQDAG)

The method that we propose extends a key component used in the top-down enumeration approach: the logical query dag (LQDAG). The LQDAG is a directed acyclic graph data structure used to represent and generate the logical plan space. It was introduced in [27] and improved in [25] by the same authors. It is also well described as the AND-OR-DAG in [41], [46] where it is used for detecting and unifying common subexpressions for multi-query optimization [41]; and for generating the space of cross-product free join trees [46].

The LQDAG contains nodes of two different types: *equivalence* nodes and *operation* nodes. Equivalence nodes can only have operation nodes as children and vice versa: operation nodes can only have equivalence nodes as children. The purpose of an equivalence node is to explicitly regroup equivalent subterms. An operation node corresponds to an algebraic operation like: join ($\bowtie$), filter (σ_θ) etc. The LQDAG can be seen as a factorized representation of a set of terms.

Inspired from [41], Figure 1 illustrates a sample LQDAG and its expansion obtained after the application of commutativity and associativity rules for the join operator.

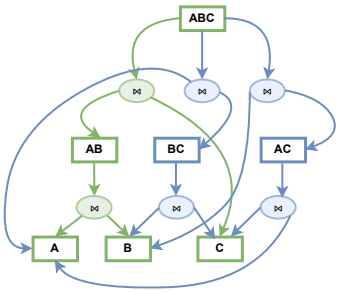

Fig. 1. Sample LQDAG (in green) with expansion (in blue).

III. The RLQDAG

The RLQDAG extends the LQDAG with the ability to capture and transform sets of recursive terms.

A. Syntax

First of all, we propose a syntax for the RLQDAG. The purpose is to be able to syntactically express a term that denotes a (potentially very large) set of recursive algebraic terms. This makes it possible to develop transformations of sets of terms formally (i.e. with a high level of precision), and express them as rewrite rules that transform one RLQDAG term d into another RLQDAG term d'. The syntax of RLQDAG terms, given in Fig. 2, focuses on formalizing the concepts of equivalence nodes, operation nodes, sharing of common subterms, and recursion.

γ	::=		Pointer to equivalence node
		$\lceil\alpha\rceil$	Equivalence node
	|	Y	Reference
α	::=		Equivalence node internals
		d	Operation node
	|	d, α	Operation nodes
	|	let Y = α_1 in α_2	Reference binder
d	::=		Operation node
		X	Relation variable
	|	$\rho_a^b(\gamma)$	Rename
	|	$\sigma_f(\gamma)$	Filter
	|	$\gamma \bowtie \gamma'$	Join
	|	$\gamma \triangleright \gamma'$	Antijoin
	|	$\gamma \cup \gamma'$	Union
	|	$\widetilde{\pi}_a(\gamma)$	Antiprojection
	|	$\mu X.\, \gamma \cup \alpha_{rec}$	Fixpoint (recursion)
α_{rec}	::=	$\lceil\alpha\rceil_{\mathfrak{D}}^{\mathfrak{R}}$	Annotated equivalence node

Fig. 2. Syntax of RLQDAG terms.

An equivalence node is a node that can have several operation nodes d as children, possibly with binders. The binder construct "let $Y = \alpha_1$ in α_2" enables the explicit sharing of a common equivalence node α_1 within the branches of another equivalence node α_2. For that purpose, it assigns a new fresh reference name Y to α_1, and allows Y to be used multiple times in α_2 as a reference to α_1. Hence, the general definition of γ is either an equivalence node $\lceil\alpha\rceil$ or a reference Y to an existing equivalence node.

Operation nodes are defined by the variable d in the abstract syntax of Fig. 2. They include the main algebraic operations of recursive relational algebra. Each operand of an operation node d points in turn to an equivalence node (through γ). The rename operator $\rho_{\mathsf{a}}^{\mathsf{b}}(\gamma)$ renames column a into column b in the equivalence node γ. The filter operator $\sigma_f(\gamma)$ applies the filtering expression f to the equivalence node γ. The antiprojection operator $\widetilde{\pi}_a(\gamma)$ removes column a from the equivalence node γ.

For example, with this syntax, the LQDAG of Fig. 1 is written as the term $[[A \bowtie B] \bowtie C]$ before expansion and is written $[[A \bowtie B] \bowtie C, A \bowtie [B \bowtie C], B \bowtie [A \bowtie C]]$ after expansion, where for the sake of readability we omit brackets around relation variables.

Recursive RLQDAGs can be expressed using fixpoint operation nodes. The principle, inspired from earlier works in recursive relational algebras [5], [32], consists in the introduction of a least-fixpoint binder operation node (μ) that binds a fresh variable X to some expression in which the variable X can appear, thus explicitly denoting recursion. In the RLQDAG, as defined in the abstract syntax of Fig. 2 and illustrated in Fig. 3, a fixpoint operator node is written $\mu X.\ \gamma\ \cup\ \alpha_{rec}$. The first

operand γ is an equivalence node that models the constant part (the base case) of the recursion. X cannot occur within γ. The other equivalence node α_{rec} is the recursive part. An important aspect is that α_{rec} contains at least one free occurrence of the recursive variable X. This is a major difference between the fixpoint operation node and the other operation nodes. This aspect will lead to a number of new definitions and formal developments. This is because depending on how the recursive variable is used in that branch, the transformation and sharing of subterms in the RLQDAG may, or may not, be allowed.

For example, on the Yago graph dataset [62], the query $\mathcal{Q}_{e_1}$:

$$?s,\ ?t \ \leftarrow\ ?s\ \texttt{isLocatedIn+}\ ?t$$

retrieves all pairs (s, t) of source and target nodes connected by a path composed of a sequence of edges labeled `isLocatedIn` (transitive closure). The following RLQDAG term Σ corresponds to query $\mathcal{Q}_{e_1}$:

$$\mu X.\ [\texttt{isLocIn}]\ \cup\ [\widetilde{\pi}_m[\rho_t^m([\texttt{isLocIn}]) \bowtie \rho_s^m([X])]]_{\mathfrak{D}}^{\mathfrak{R}}$$

It makes recursion explicit using the fixpoint operator. The equivalence node for the constant part contains the relation variable `isLocIn` whose column names are s and t. The equivalence node for the recursive part is composed of a join between the recursive variable X with the relation variable `isLocIn`. Here, the path traversal is performed from right to left, by introducing a temporary column name "m" and renaming the columns so that the natural join is performed on the only common column "m" before "m" is discarded by the antiprojection.

Fig. 3 illustrates the RLQDAG of $\mathcal{Q}_{e_1}$ with two recursive subterms Σ and Σ'. Notice that Σ' is semantically equivalent to Σ and encodes the left to right direction of traversal using a different column renaming in the recursive part.

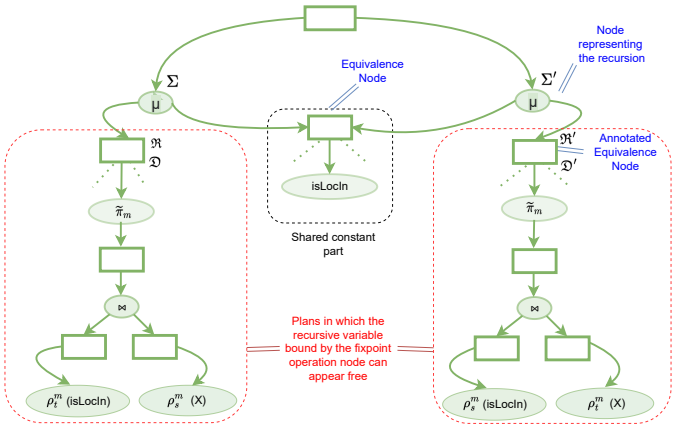

Fig. 3. Structure of recursive terms in RLQDAG.

In a RLQDAG, the recursive equivalence node of each fixpoint operation node is annotated with annotations $\mathfrak{D}$ and $\mathfrak{R}$. These annotations will be useful for guiding the application of transformations and will be defined and explained in Section III-D.

There are two reasons why we distinguish α_{rec} from a general equivalence node α in the abstract syntax. The first reason is that those equivalence nodes for recursive parts are equipped with annotations (see Section III-E.). The second reason is that we want to allow a maximum level of sharing while preventing the sharing of subterms with free occurences of a recursive variable. We thus forbid the use of the binding construct to share subterms with free variables.

We consider the following restrictions over the abstract syntax presented in Fig. 2: we consider only positive, linear and non mutually recursive RLQDAG terms: (i) positive means that recursive variables only appear in the left-hand operand of an antijoin operator node; (ii) linear means that one of the operands in a join or antijoin operator node is constant in the free variable; (iii) Non-mutually recursive terms means that fixpoint operator nodes are properly nested so that there is only one free variable in any subbranch of an annotated equivalence node (this variable may occur several times). These restrictions define a subset of RLQDAG terms that simplifies the theory while supporting the encoding of expressive queries like union of conjunctive regular path queries (UCRPQs).

B. Semantics of RLQDAG terms

The interpretation of a RLQDAG term is the set of all recursive relational algebraic terms that it represents. Formally, the semantics of a RLQDAG $[\alpha]$ is a set of μ-RA terms as defined by the functions $S_\alpha[\![]\!]$ and $S_\gamma[\![]\!]$ presented in Fig. 4, where E denotes a variable environment used to keep track of the variable definitions introduced by binders for the sharing of subterms. The interpretation of a RLQDAG $[\alpha]$ is $S_\alpha[\![\alpha]\!]_\emptyset$.

$$\begin{aligned}
S_\gamma[\![[\alpha]]\!]_E &= S_\alpha[\![\alpha]\!]_E \\
S_\gamma[\![Y]\!]_E &= S_\alpha[\![E(Y)]\!]_E \\
\\
S_\alpha[\![d]\!]_E &= S_d[\![d]\!] \\
S_\alpha[\![d, \alpha]\!]_E &= S_d[\![d]\!] \cup S_\alpha[\![\alpha]\!]_E \\
S_\alpha[\![\text{let } Y = \alpha_1 \text{ in } \alpha_2]\!]_E &= S_\alpha[\![\alpha_2]\!]_{E \oplus \{Y \mapsto \alpha_1\}} \\
\\
S_d[\![X]\!] &= X \\
S_d[\![\sigma_f(\gamma)]\!] &= \{\ \sigma_f(t) \mid t \in S_\gamma[\![\gamma]\!]_E\} \\
S_d[\![\gamma_1 \bowtie \gamma_2]\!] &= \{\ t \bowtie t' \mid t \in S_\gamma[\![\gamma_1]\!]_E \wedge \\
&\qquad t' \in S_\gamma[\![\gamma_2]\!]_E\} \\
S_d[\![\gamma_1 \triangleright \gamma_2]\!] &= \{\ t \triangleright t' \mid t \in S_\gamma[\![\gamma_1]\!]_E \wedge \\
&\qquad t' \in S_\gamma[\![\gamma_2]\!]_E\} \\
S_d[\![\gamma_1 \cup \gamma_2]\!] &= \{\ t \cup t' \mid t \in S_\gamma[\![\gamma_1]\!]_E \wedge \\
&\qquad t' \in S_\gamma[\![\gamma_2]\!]_E\} \\
S_d[\![\rho_a^b(\gamma)]\!] &= \{\ \rho_a^b(t) \mid t \in S_\gamma[\![\gamma]\!]_E\} \\
S_d[\![\widetilde{\pi}_a(\gamma)]\!] &= \{\ \widetilde{\pi}_a(t) \mid t \in S_\gamma[\![\gamma]\!]_E\} \\
S_d[\![\mu X.\ \gamma \cup \alpha_{rec}]\!] &= \{\mu X.\ t \cup t_{rec} \mid t \in S_\gamma[\![\gamma]\!]_E \\
&\qquad \wedge\ t_{rec} \in S_\alpha[\![\alpha_{rec}]\!]_E\}
\end{aligned}$$

Fig. 4. Formal semantics of RLQDAG terms, with one interpretation function for each syntactic construct of Fig. 2.

A well-formed RLQDAG is a RLQDAG whose interpretation is a set of semantically equivalent terms:

Definition 1 (Well-formedness). *A RLQDAG $[\alpha]$ is well-formed if and only if* $\forall t, t' \in S_\alpha[\![\alpha]\!]_\emptyset,\quad \texttt{eval}(t) = \texttt{eval}(t')$.

In this definition $\texttt{eval}(t)$ returns the interpretation of the individual recursive relational algebraic term t (i.e. the set of tuples returned by t when evaluated in a database instance).

For example, Fig. 5 illustrates a well-formed RLQDAG capturing two semantically equivalent relational terms obtained before and after the application of join distributivity over union. Fig. 6 illustrates a RLQDAG which is not well-formed, since its top-level equivalence node contains two subterms that are not semantically equivalent.

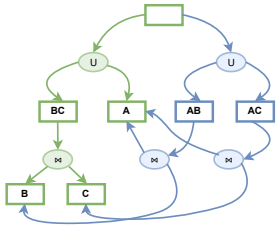

Fig. 5. Well-formed RLQDAG.

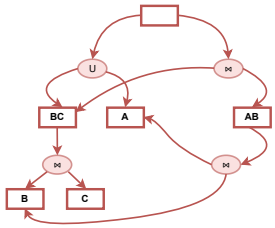

Fig. 6. Not well-formed RLQDAG.

The *type* of an operation node d is the set of column names obtained in the result of the evaluation of any subbranch of d. In a well-formed RLQDAG, all d_i under the same equivalence node are semantically equivalent, and thus have the same type. For this reason, we also define the *type of an equivalence node*: $type(\gamma)$ as the type of one of its operation nodes. Notice that the type of an annotated equivalence node of some fixpoint operation node d corresponds to the type of the equivalence node of the constant part of d. For a given filter operation node $\sigma_f(\gamma)$, we denote by $filt(f) \subseteq type(\sigma_f(\gamma))$ the subset of column names used in the filtering function f.

C. Recursive terms and rule applicability

A significant novelty introduced by recursion when compared to the classical setting of non-recursive algebraic terms resides in the criteria used to trigger rewrite rules. In the classical non-recursive setting these criteria are trivial in the sense that they only depend on top-level operators. In the example of Fig. 5, when applying join distributivity over union, the applicability of the rewrite rule $A \cup (B \bowtie C) \longrightarrow (A \bowtie B) \cup (A \bowtie C)$ can be determined by examining only the combination of the two top-most operators, i.e. the top-level ($\cup$) with the operator immediately underneath (i.e. $\bowtie$).

For some other rewrite rules, applicability criteria may also include some additional verifications such as (non)-interaction between e.g. the set of columns being filtered, or the columns being removed (in the cases of filter and antiprojection, respectively). In the non-recursive setting, these verifications can be done without the need to traverse the whole term. This means that usually no further traversal of subtrees of operators are required. For instance, in the previous example of join distributivity over union, B and C do not need to be traversed at all when determining rule applicability. In sharp contrast, rules for transforming recursive terms rely on criteria that are significantly more sophisticated as they sometimes require a whole traversal of the recursive part of a fixpoint term. This is because opportunities for rule application with recursive terms depend on how the recursive variable is used within the recursive parts of fixpoints. It is known since the works of [5], [32], [33] that criteria for rule application are significantly more sophisticated in the presence of recursion as they need to examine how the recursive variables are used. We show that it is still possible to apply rules over sets of recursive terms at once, using the concept of *annotated equivalence nodes*, that we define in § III-E, and for which we need some preliminary definitions.

D. Preliminary definitions of auxiliary functions in RLQDAG

Definition 2 (Unfolding)**.** *Let α be an equivalence node. The unfolding of α, denoted $\texttt{unfold}(\alpha)$ is α in which all occurrences of equivalence node variable names Y are replaced by their definitions (binders are simply unfolded).*

We now define two auxiliary functions over RLQDAG terms. These functions will be used for defining annotated equivalence nodes in Section III-E.

a) Notion of destabilizer in RLQDAG:
We define the destabilizer of an RLQDAG equivalence node as the set of columns that can be modified by an iteration of a parent fixpoint node. Specifically, $\texttt{destab}()$ traverses subterms and analyzes how the occurrences of free variables are used in order to compute the set of columns that are subject to modifications during a fixpoint node iteration (e.g. renaming or antiprojection).

Definition 3. *For a fixpoint operation node $\mu X.\ \gamma \cup \alpha_{rec}$ we consider $\alpha' = \texttt{unfold}(\alpha_{rec})$ and we define $\texttt{destab}(\alpha', X)$ as the following set of column names: $\texttt{destab}(\alpha', X) = \{c \in \mathfrak{C} \mid \exists p \in d(\alpha', X)\ \ p(c) \neq c\}$ where $d(\cdot,\cdot)$ computes the set of derivations over a RLQDAG term:*

$$
\begin{aligned}
d((d,\alpha),X) &= d(d,X)\\
d([\alpha_1]\cup[\alpha_2],X) &= d(\alpha_1,X)\cup d(\alpha_2,X)\\
d([\alpha_1]\triangleright[\alpha_2],X) &= d(\alpha_1,X)\\
d([\alpha_1]\bowtie[\alpha_2],X) &= d(\alpha_1,X)\cup d(\alpha_2,X)\\
d(\rho_a^b([\alpha]),X) &= \{p\circ(b\to a, a\to\perp)\mid p\in d(\alpha,X)\}\\
d(\widetilde{\pi}_a([\alpha]),X) &= \{p\circ(a\to\perp)\mid p\in d(\alpha,X)\}\\
d(\sigma_f([\alpha]),X) &= d(\alpha,X)\\
d(\mu(Z.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}),X) &= \emptyset\\
d(X,X) &= \{()\}\ \textit{(a singleton identity)}\\
d(R,X) &= \emptyset
\end{aligned}
$$

and where $\circ$ represents the composition and $(a_1 \to b_1, ..., a_n \to b_n)$ denotes the function that maps each a_i to its b_i and every other column name to itself.

For instance, in the RLQDAG of Fig. 3, $\mathfrak{D} = \{s, m\}$ in Σ and $\mathfrak{D}' = \{t, m\}$ in Σ'. Intuitively, this is because these columns are renamed in front of the recursive variables.

b) Notion of rigidity in RLQDAG: We define a function $\texttt{rigid}()$ that computes the set of columns that cannot be added nor removed from a fixpoint operation node, without breaking the semantics of the RLQDAG term. A column $c \in \mathfrak{C}$ cannot be added nor removed from an annotated equivalence node α_{rec} (recursive in X) when

$c \in \texttt{rigid}(\texttt{unfold}(\alpha), X)$ and rigid() is defined as follows:

Definition 4 (Rigidity)**.**

$$
\begin{aligned}
\texttt{rigid}((d,\alpha),X) &= \texttt{rigid}(d,X)\\
\texttt{rigid}(\lceil\alpha_1\rceil\cup\lceil\alpha_2\rceil,X) &= \texttt{rigid}(\alpha_1,X)\cup\texttt{rigid}(\alpha_2,X)\\
\texttt{rigid}(\lceil\alpha_1\rceil\bowtie\lceil\alpha_2\rceil,X) &= \texttt{rigid}(\alpha_1,X)\cup\texttt{rigid}(\alpha_2,X)\\
\texttt{rigid}(\lceil\alpha_1\rceil\triangleright\lceil\alpha_2\rceil,X) &= \texttt{rigid}(\alpha_1,X)\cup\texttt{rigid}(\alpha_2,X)\\
\texttt{rigid}(\rho^b_a(\lceil\alpha\rceil),X) &= \texttt{rigid}(\alpha,X)\cup\{a,b\}\\
\texttt{rigid}(\widetilde{\pi}_a(\lceil\alpha\rceil),X) &= \emptyset \ \textit{ when } X\notin \text{free}(\alpha)\\
&= \texttt{rigid}(\alpha,X)\cup\{a\} \ \textit{otherwise}\\
\texttt{rigid}(\sigma_f(\lceil\alpha\rceil),X) &= \texttt{rigid}(\alpha,X)\cup\text{filt}(f)\\
\texttt{rigid}(\mu(Z.\gamma\cup\lceil\alpha\rceil^{\mathfrak{R}}_{\mathfrak{D}}),X) &= \texttt{rigid}(\alpha,X)\cup\texttt{rigid}(\gamma,X)\\
\texttt{rigid}(R,X) &= \textit{type}(R) \textit{ when } X\neq R\\
\texttt{rigid}(X,X) &= \emptyset
\end{aligned}
$$

For instance, in the RLQDAG of Fig. 3, $\mathfrak{R}=\{s,t\}$ in Σ. Intuitively, this is because these columns cannot be added nor removed without changing the semantics of the recursion.

E. Annotated equivalence node

An annotated equivalence node (α_{rec} in the abstract syntax of RLQDAG terms given in Fig. 2) is an equivalence node of a recursive part of a fixpoint, which is annotated with information that characterize how the recursive variable is used. Specifically:

Definition 5. *Given a RLQDAG operation node $d = \mu X.\ \gamma\cup\alpha_{rec}$, the annotated equivalence node α_{rec} is defined as: $\lceil\alpha\rceil^{\mathfrak{R}}_{\mathfrak{D}}$ where $\mathfrak{D}=\texttt{destab}(\alpha,X)$ and $\mathfrak{R}=\texttt{rigid}(\alpha,X)$.*

For example, in the RLQDAG of Fig. 3, the two subterms Σ and Σ' (that correspond to the right-to-left and left-to-right path traversals) carry different annotations: $\mathfrak{D}=\{s,m\}$ in Σ whereas $\mathfrak{D}'=\{t,m\}$ in Σ'.

Annotations are intended to characterize all the subterms of the annotated equivalence node (thanks to definition 6). The goal of annotated equivalence nodes is to guide and maximize the grouped application of transformations, while also maximizing the sharing of common subterms. In the sequel, we detail RLQDAG transformations, by introducing new rewrite rules capable of transforming sets of subterms at once.

a) Notion of consistency: Intuitively, a RLQDAG is *consistent* iff it is well-formed and in addition, for any of its annotated equivalence nodes $\lceil\alpha_2\rceil^{\mathfrak{R}}_{\mathfrak{D}}$, the annotations $\mathfrak{D}$ and $\mathfrak{R}$ are the same for all operation nodes directly underneath (no matter on which subbranch of the equivalence node they are computed, they coincide). More formally:

Definition 6 (Consistency)**.** *A RLQDAG $\lceil\alpha\rceil$ is consistent iff:*

1) *it is well-formed;*
2) *for all fixpoint operator node $\mu X.\gamma\cup\lceil\alpha_2\rceil^{\mathfrak{R}}_{\mathfrak{D}}$ occurring in α, we have $\texttt{cons}(\alpha_2,X)^{\mathfrak{R}}_{\mathfrak{D}}$ where:*

$$
\begin{aligned}
\texttt{cons}(\lceil d\rceil,X)^{\mathfrak{R}}_{\mathfrak{D}} &= \texttt{destab}(d,X)=\mathfrak{D}\\
&\quad\wedge\ \texttt{rigid}(d,X)=\mathfrak{R}\\
\texttt{cons}(\lceil d,\alpha\rceil,X)^{\mathfrak{R}}_{\mathfrak{D}} &= \texttt{destab}(d,X)=\mathfrak{D}\\
&\quad\wedge\ \texttt{rigid}(d,X)=\mathfrak{R}\\
&\quad\wedge\ \texttt{cons}(\lceil\alpha\rceil,X)^{\mathfrak{R}}_{\mathfrak{D}}
\end{aligned}
$$

An example of a consistent RLQDAG is shown in Fig. 3 provided $\mathfrak{D},\mathfrak{R},\mathfrak{D}'$ and $\mathfrak{R}'$ are correctly computed. An example of an inconsistent RLQDAG would be a variant of Fig. 3 with a wrong annotation (e.g. $\mathfrak{D}'=\mathfrak{D}$), later exposing the structure to incorrect transformations. This is because incorrect criteria satisfaction might then result in, for example, wrongly pushing a join operation node inside a fixpoint operation node. This would typically produce a not well-formed RLQDAG, breaking the semantics of the initial query. In the remaining, we only consider consistent RLQDAGs.

IV. RLQDAG Transformations

We now propose RLQDAG transformations whose purpose is to efficiently build the space of equivalent recursive plans.

RLQDAG transformations capture all the most advanced rewritings of recursive algebraic terms found in previous approaches (e.g. [4], [32]). Unlike previous approaches RLQDAG transformations make it possible to systematically group sets of recursive terms and exploit the sharing of common subterms to avoid redundant computations.

RLQDAG transformations leverage annotated equivalence nodes to guide the transformation of recursive subterms. They also update the RLQDAG structure with new annotations when needed, in an incremental manner. The incremental aspect for updating annotations is important as it avoids numerous subterm traversals, thus enabling more efficient grouped transformations.

For each recursive transformation, we describe which subterms can be shared, how newly generated terms are attached and what happens with the other plans already present in the equivalence node. The creation of new combinations of operation nodes may in turn generate more opportunities for transformations that are also explored.

A. RLQDAG rewrite rules

We formalize all these ideas by introducing RLQDAG rewrite rules, based on the syntax of RLQDAG terms introduced in Section III. Specifically, RLQDAG rewrite rules are formalized as functions that take an equivalence node γ and return another equivalence node γ' obtained after applying transformations.

a) Pushing filters into fixpoint operation nodes: For pushing filters into sets of recursive terms, we introduce a function pf(), defined by considering all the syntactic decomposition cases of the input γ. Fig. 7 focuses on the two main cases that correspond to potential opportunities of pushing filters, i.e. the cases $\text{pf}(\lceil d\rceil)$ and $\text{pf}(\lceil d,\alpha\rceil)$ where d filters an equivalence node which contains a recursive subterm. For all the other cases, pf() does not reorder operation nodes in the RLQDAG structure but simply traverses it in search for further transformation opportunities underneath[1].

[1]For instance, two sample cases are the following:

$$
\begin{aligned}
\text{pf}(\mu X.\ \gamma\ \cup\ \lceil\alpha\rceil^{\mathfrak{R}}_{\mathfrak{D}}) &= \mu X.\ \texttt{expand}(\gamma)\ \cup\ \lceil\texttt{expand}(\alpha)\rceil^{\mathfrak{R}}_{\mathfrak{D}}\\
\text{pf}(\lceil[A]\ \bowtie\ [B]\rceil) &= \texttt{expand}(\lceil A\rceil)\ \bowtie\ \texttt{expand}(\lceil B\rceil)
\end{aligned}
$$

$$\text{pf}([\sigma_f([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}])]) =$$
$$\begin{cases} \bullet\ [\mu X'.\ \text{expand}([\sigma_f(\gamma)])\ \cup\ [\text{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}'}_{\mathfrak{D}}\ ] \\ \quad \textit{when}\ \ \textit{filt(f)} \cap \mathfrak{D} = \emptyset \\ \bullet\ [\ \sigma_f(\text{expand}([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}]))\ ]\ \ \textit{otherwise} \end{cases}$$

$$\text{pf}([\sigma_f([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\ \alpha_2])]) =$$
$$\begin{cases} \bullet\ [\mu X'.\ \text{expand}([\sigma_f(\gamma)])\ \cup\ [\text{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}'}_{\mathfrak{D}}, \\ \quad \text{expand}([\sigma_f([\alpha_2])])\ ] \\ \quad \textit{when}\ \ \textit{filt(f)} \cap \mathfrak{D} = \emptyset \\ \bullet\ [\ \sigma_f(\text{expand}([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\ \alpha_2]))\ , \\ \quad \text{expand}([\sigma_f([\alpha_2])])\ ]\ \ \textit{otherwise} \end{cases}$$

where $\mathfrak{R}' = \mathfrak{R} \cup \textit{filt(f)}$ and $\alpha_{\{X/X'\}}$ denotes α in which all occurrences of X are replaced by X'.

Fig. 7. Pushing a filter in an equivalence node containing a fixpoint.

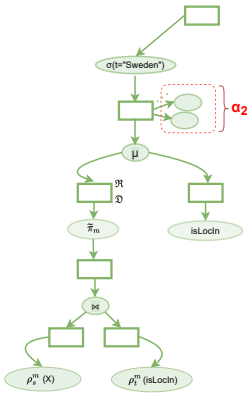

Fig. 8. Initial RLQDAG before expansion with opportunity to apply pf().

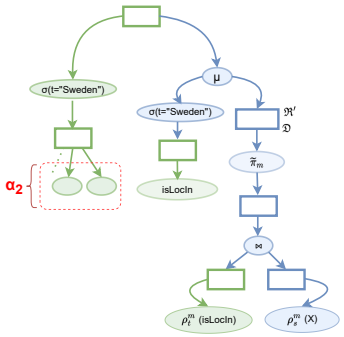

Fig. 9. Expansion of a RLQDAG by pushing a filter in a fixpoint operation node, with branches added by pf() in blue color.

In Fig. 7, whenever the filter can be pushed through the fixpoint operation node, pf() generates a new RLQDAG in which the filter operation node is put within the constant part of the fixpoint operation node. In the resulting equivalence node, the initial (suboptimal) term is replaced by the new term where filtering is done earlier (hence more efficient). For this transformation to happen, the criteria (on the last line of Fig. 7) must be satisfied. Whenever it is not the case, the RLQDAG structure is not reordered but simply recursively traversed in search for more transformation opportunites. The creation of new combinations of operation nodes may in turn provide new opportunities for other rewritings (for instance, new opportunities for pushing filters even further, or even other kinds of rewritings). This is the role of the `expand()` function, formally defined in section IV-B. Intuitively, a call to `expand()` on an equivalence node may further populate the equivalence node with new subterms. The `expand()` function is in charge of exploring all opportunities for transformations. This is useful because other rewrite rules may apply, and the `expand()` function basically triggers all possible applications of all rewrite rules.

For example, the query $\mathcal{Q}_{e_2}$:

$$?s \ \leftarrow\ ?s\ \texttt{isLocatedIn+}\ \text{Sweden}$$

retrieves all nodes that are connected to the constant node "Sweden" by a path composed of a sequence of edges labeled `isLocatedIn` from the Yago graph dataset [62]. Fig. 8 illustrates graphically a portion of $\mathcal{Q}_{e_2}$ RLQDAG, and Fig. 9 depicts its updated structure obtained after the pushing filter transformation defined in Fig. 7. New branches created by pf() are represented in blue color. The new term is added in the same equivalence node as the previous term, since they are semantically equivalent. Notice the incremental update of annotations performed by pf() in Fig. 7: the annotations of the newly created term (in blue) are obtained from the annotations of the initial term. In that case $\mathfrak{D}$ is simply propagated whereas $\mathfrak{R}' = \mathfrak{R} \cup \textit{filt(f)}$.

b) Pushing joins into fixpoint operation nodes: For pushing joins into sets of recursive terms we define a function pj(). pj() takes an equivalence node γ as input and returns an expanded equivalence node γ' that contains all the subterms in γ with, in addition, all the subterms where all joins pushable in fixpoint operation nodes have been pushed. We define pj() for all possible syntactic decompositions of a RLQDAG. Fig. 10 presents the definition of pj() for the two main cases of interest. Again, other cases are defined without structure rearrangement but involving recursive calls to `expand()` in search for further transformation opportunites underneath. Notice that whenever a join can be pushed within a fixpoint operation node (see Fig. 10), it is possible to share the constant part of the fixpoint operation node. This is made explicit by the creation of the outermost "let" binder whose goal is to define and associate a name to the equivalence node, so that it can be referred multiple times (thus explicitly showing the sharing of subterms).

For example, the query $\mathcal{Q}_{e_3}$:

$$?s, ?t\ \leftarrow\ ?s\ \texttt{haschild+/livesin}\ ?t$$

retrieves all pairs of nodes that are connected by a path composed of a sequence of edges labeled `haschild` followed by a single edge labeled `livesin`. Fig. 11 illustrates graphically a portion of $\mathcal{Q}_{e_3}$ RLQDAG, after the application of the rule that pushes joins into fixpoint operation nodes described in Fig. 10. Fig. 11 shows that the newly created branch (in blue color) extends the set of semantically equivalent terms of the existing equivalence node.

c) Merging fixpoint operation nodes: The function mf() defined in Fig. 12 takes an input γ and returns an equivalence node γ' containing all the subterms in γ with, in addition, all the terms in which recursions that can be merged are merged. A merging happens whenever (i) two recursions are joined and (ii) their annotated equivalence nodes allow them to be merged into a single recursion, as described in Fig. 12. The

where the `expand()` function, defined in Section IV-B, simply traverses subterms in search for more transformation opportunities. pf() is defined similarly for all the other syntactic cases of γ. Notice that when γ is a reference Y, there is no need to introduce a new binder, the reference name is used directly.

$$\text{pj}([\beta \bowtie [\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}]]) =$$

$$\begin{cases} \bullet\ \text{let const} = \gamma \text{ in} \\ \quad [\ [\texttt{expand}(\beta) \bowtie \texttt{expand}([\mu X.\ \text{const}\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}])]\ , \\ \quad \mu X'.\ \texttt{expand}([\beta \bowtie [\text{const}]])\ \cup\ [\texttt{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}'}_{\mathfrak{D}'}\ ] \\ \quad \textit{when}\ \ \textit{type}(\beta) \cap \mathfrak{D} = \emptyset\ \textit{and}\ \textit{type}(\beta) \backslash \textit{type}(\gamma)\ \cap \mathfrak{R} = \emptyset \\ \bullet\ [\texttt{expand}(\beta) \bowtie \texttt{expand}([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}])] \qquad \textit{otherwise} \end{cases}$$

$$\text{pj}([\beta \bowtie [\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\ \alpha_2]]) =$$

$$\begin{cases} \bullet\ \text{let const} = \gamma \text{ in} \\ \quad [\ [\texttt{expand}(\beta) \bowtie \texttt{expand}([\mu X.\ \text{const}\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\ \alpha_2])]\ , \\ \quad \mu X'.\ \texttt{expand}([\beta \bowtie [\text{const}]])\ \cup\ [\texttt{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}'}_{\mathfrak{D}'}\ , \\ \quad [\texttt{expand}(\beta) \bowtie \texttt{expand}([\alpha_2])]\ ] \\ \quad \textit{when}\ \ \textit{type}(\beta) \cap \mathfrak{D} = \emptyset\ \textit{and}\ \textit{type}(\beta) \backslash \textit{type}(\gamma)\ \cap \mathfrak{R} = \emptyset \\ \bullet\ [[\texttt{expand}(\beta) \bowtie \texttt{expand}([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\ \alpha_2])]\ , \\ \quad [\texttt{expand}(\beta) \bowtie \texttt{expand}([\alpha_2])]\ ] \qquad \textit{otherwise} \end{cases}$$

where we define $\mathfrak{D}' = \mathfrak{D} \cup \texttt{destab}(\beta, X)$ and $\mathfrak{R}' = \mathfrak{R} \cup \texttt{rigid}(\beta, X)$. This is because we need to recompute the destabilizer on the underlying set of derivations updated with $\texttt{d}(\beta, X)$ and we have $\texttt{destab}(A \cup B) = \texttt{destab}(A) \cup \texttt{destab}(B)$.

Fig. 10. Pushing join in an equivalence node containing a fixpoint.

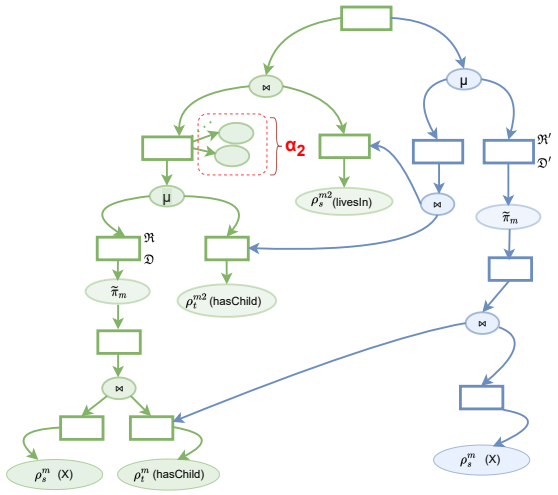

Fig. 11. Expansion of a RLQDAG by pushing a join in a fixpoint operation node. The initial RLQDAG is in green color, and parts added by pj() are in blue color.

constant part of the new recursive term created is the join of the constant parts of the two initial fixpoints, and a new recursive part is created. Since the constant part has changed, a new recursive variable is introduced and recursive parts are also new (and cannot be shared[2]).

For example, the query $\mathcal{Q}_{e_4}$:

$$?s,\ ?t\ \leftarrow\ ?s\ \texttt{islocatedin+/dealswith+}\ ?t$$

retrieves all pairs of nodes that are connected by a path composed of two successive sequences of edges labeled `islocatedin` and `dealswith` respectively. Fig. 13 illustrates a portion of $\mathcal{Q}_{e_4}$ RLQDAG, with the new recursive term produced, using subterm-sharing, after the merging of fixpoint operation nodes.

d) Pushing an antiprojection in a fixpoint operation node: For pushing antiprojections into fixpoint operation nodes we introduce a function pp() that takes an equivalence node γ as input and returns an expanded equivalence node γ' where all pushable antiprojections have been pushed. pp() is defined in Fig. 14. The antiprojection is pushed when the criteria allows it, and this results in the creation of a new term which replaces the initial one in the existing equivalence node. Whenever the criteria is not satisfied, the initial term is left unchanged, but traversed in search for more transformation opportunities.

e) Pushing an antijoin in a fixpoint operation node: For pushing antijoins into fixpoints we introduce a function pa() that takes an equivalence node γ as input and returns an expanded node γ' that contains all the subterms in γ with, in addition, all subterms where all pushable antijoins have been pushed in fixpoint operation nodes. pa() is defined in Fig. 15. Again, Fig. 15 focuses on the syntactic cases that correspond to when an antijoin might be pushed in a fixpoint. When criteria are satisfied, the antijoin is pushed in the constant part of the fixpoint operation node and the newly created term is added to the set of equivalent terms along with the rest.

RLQDAG rules can be divided into two groups. The first group is composed of rewrite rules pj(), pa() and mf(). The application of these rules always preserve preexisting terms in an equivalence node. These rules can only perform additions of new operation nodes in an equivalence node[3]. The second group is composed of the rules pf() and pp() that perform logical optimizations. The application of these rules can replace terms in an equivalence node by more optimal variants (i.e. they discard logically suboptimal variants).

B. The overall expansion algorithm

We can now describe the overall RLQDAG expansion algorithm. We define the function `expand()` that takes an equivalence node γ and returns the equivalence node γ' which contains all the terms obtained by transformations. The expand function is simply defined as follows:

$$\begin{aligned} \texttt{expand}([d]) &= \texttt{applyAll}([d]) \\ \texttt{expand}([d, \alpha]) &= \texttt{applyAll}([d]) \cup \texttt{expand}(\alpha) \end{aligned}$$

where `applyAll()` is in charge of applying all possible transformations on each operation node. This includes applying all rewrite rules defined in Section IV in combination with the more classical ones of relational algebra[4]:

$$\begin{aligned} \texttt{applyAll}([d]) = \ & \text{pf}([d]) \cup \text{pa}([d]) \cup \text{pj}([d]) \cup \text{mf}([d]) \\ & \cup\ \text{pp}([d]) \cup \texttt{allCodd}([d]) \end{aligned}$$

[2]This does not prevent the sharing of potential subparts with no occurrence of a free variable.

[3]One operation node will then be selected using cost estimation heuristics depending on statistics on the cardinalities of the database instance.

[4]As a slight discrepancy between the theory and the implementation, the implementation of the RLQDAG expansion avoids some redundant calls to the `expand()` function by implementing the `applyAll()` function in an imperative-style double nested for loop, so as to implement a single case analysis and to trigger `expand()` once only when needed.

$$\text{mf}([\ [\mu X_1.\ \gamma_1\ \cup\ [\alpha_1]^{\mathfrak{R}_1}_{\mathfrak{D}_1}] \bowtie [\mu X_2.\ \gamma_2\ \cup\ [\alpha_2]^{\mathfrak{R}_2}_{\mathfrak{D}_2}]\ ]) =$$

$$\begin{cases} \bullet\ \text{let const}_1 = \gamma_1 \text{ in} \\ \quad \text{let const}_2 = \gamma_2 \text{ in} \\ \quad\quad [\ \texttt{expand}([\mu X_1.\ \text{const}_1\ \cup\ [\alpha_1]^{\mathfrak{R}_1}_{\mathfrak{D}_1}]) \bowtie \texttt{expand}([\mu X_2.\ \text{const}_2\ \cup\ [\alpha_2]^{\mathfrak{R}_2}_{\mathfrak{D}_2}])\ , \\ \quad\quad \mu X.\ \texttt{expand}([\text{const}_1 \bowtie \text{const}_2]) \cup \texttt{expand}([\alpha_{1\{X_1/X\}}\ \cup\ \alpha_{2\{X_2/X\}}]^{\mathfrak{R}}_{\mathfrak{D}})\ ] \\ \quad when\ \ (type(\gamma_1) \cap type(\gamma_2)) \cap (\mathfrak{D}_1 \cup \mathfrak{D}_2) = \emptyset\ and\ type(\gamma_1) \backslash type(\gamma_2) \cap \mathfrak{R}_2 = \emptyset\ and\ type(\gamma_2) \backslash type(\gamma_1) \cap \mathfrak{R}_1 = \emptyset \\ \bullet\ \texttt{expand}([\mu X_1.\ \text{const}_1\ \cup\ [\alpha_1]^{\mathfrak{R}_1}_{\mathfrak{D}_1}]) \bowtie \texttt{expand}([\mu X_2.\ \text{const}_2\ \cup\ [\alpha_2]^{\mathfrak{R}_2}_{\mathfrak{D}_2}])\quad otherwise \end{cases}$$

$$\text{mf}([\ [\mu X_1.\ \gamma_1\ \cup\ [\alpha_1]^{\mathfrak{R}_1}_{\mathfrak{D}_1}, \alpha_3] \bowtie [\mu X_2.\ \gamma_2\ \cup\ [\alpha_2]^{\mathfrak{R}_2}_{\mathfrak{D}_2}, \alpha_4]\ ]) =$$

$$\begin{cases} \bullet\ \text{let const}_1 = \gamma_1 \text{ in} \\ \quad \text{let const}_2 = \gamma_2 \text{ in} \\ \quad\quad [\ \texttt{expand}([\mu X_1.\ \text{const}_1\ \cup\ [\alpha_1]^{\mathfrak{R}_1}_{\mathfrak{D}_1}]) \bowtie \texttt{expand}([\mu X_2.\ \text{const}_2\ \cup\ [\alpha_2]^{\mathfrak{R}_2}_{\mathfrak{D}_2}])\ , \\ \quad\quad \mu X.\ \texttt{expand}([\text{const}_1 \bowtie \text{const}_2]) \cup \texttt{expand}([\alpha_{1\{X_1/X\}}\ \cup\ \alpha_{2\{X_2/X\}}]^{\mathfrak{R}}_{\mathfrak{D}})\ , \\ \quad\quad \texttt{expand}([\alpha_3]) \bowtie \texttt{expand}([\mu X_2.\ \gamma_2\ \cup\ [\alpha_2]^{\mathfrak{R}_2}_{\mathfrak{D}_2}, \alpha_4])\ , \\ \quad\quad \texttt{expand}([\alpha_4]) \bowtie \texttt{expand}([\mu X_1.\ \gamma_1\ \cup\ [\alpha_1]^{\mathfrak{R}_1}_{\mathfrak{D}_1}, \alpha_3])\ ] \\ \quad when\ \ (type(\gamma_1) \cap type(\gamma_2)) \cap (\mathfrak{D}_1 \cup \mathfrak{D}_2) = \emptyset\ \ and\ \ type(\gamma_1) \backslash type(\gamma_2) \cap \mathfrak{R}_2 = \emptyset\ \ and\ \ type(\gamma_2) \backslash type(\gamma_1) \cap \mathfrak{R}_1 = \emptyset \\ \bullet\ [\ \texttt{expand}([\mu X_1.\ \gamma_1\ \cup\ [\alpha_1]^{\mathfrak{R}_1}_{\mathfrak{D}_1}, \alpha_3]) \bowtie \texttt{expand}([\mu X_2.\ \gamma_2\ \cup\ [\alpha_2]^{\mathfrak{R}_2}_{\mathfrak{D}_2}, \alpha_4])\ , \\ \quad \texttt{expand}([\alpha_3]) \bowtie \texttt{expand}([\mu X_2.\ \gamma_2\ \cup\ [\alpha_2]^{\mathfrak{R}_2}_{\mathfrak{D}_2}, \alpha_4])\ , \\ \quad \texttt{expand}([\alpha_4]) \bowtie \texttt{expand}([\mu X_1.\ \gamma_1\ \cup\ [\alpha_1]^{\mathfrak{R}_1}_{\mathfrak{D}_1}, \alpha_3])\ ]\quad otherwise \end{cases}$$

where $\mathfrak{D} = \mathfrak{D}_1 \cup \mathfrak{D}_2$ and $\mathfrak{R} = \mathfrak{R}_1 \cup \mathfrak{R}_2$ and $\alpha_{i\{X_i/X\}}$ denotes α_i in which all occurrences of X_i are replaced by X. This is because the only transformation of the recursive part that needs to be propagated to update the annotations is the union of the two former recursive parts and both `destab` and `rigid` are distributive over union.

Fig. 12. Merging fixpoint operation nodes.

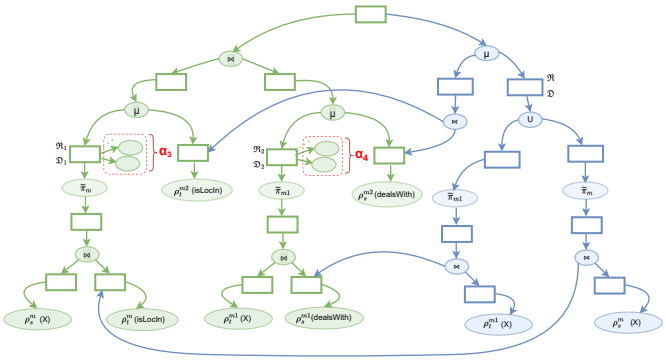

Fig. 13. RLQDAG structure after merging fixpoint operation nodes.

$$\text{pa}([[\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}] \triangleright \beta]) =$$

$$\begin{cases} \bullet\ \text{let const} = \gamma \text{ in} \\ \quad [\ [\texttt{expand}([\mu X.\ \text{const}\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}]) \triangleright \texttt{expand}(\beta)]\ , \\ \quad \mu X'.\ \texttt{expand}([\text{const} \triangleright \beta]) \cup [\texttt{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}'}_{\mathfrak{D}}\ ] \\ \quad when\ \ type(\beta) \cap \mathfrak{D} = \emptyset \\ \bullet\ [\ \texttt{expand}([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}]) \triangleright \texttt{expand}(\beta)\ ]\quad otherwise \end{cases}$$

$$\text{pa}([[\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\ \alpha_2] \triangleright \beta]) =$$

$$\begin{cases} \bullet\ \text{let const} = \gamma \text{ in} \\ \quad [\ [\texttt{expand}([\mu X.\ \text{const}\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\ \alpha_2]) \triangleright \texttt{expand}(\beta)], \\ \quad [\mu X'.\ \texttt{expand}([\text{const} \triangleright \beta]) \cup [\texttt{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}'}_{\mathfrak{D}}], \\ \quad [\texttt{expand}([\alpha_2]) \triangleright \texttt{expand}(\beta)]\ ] \\ \quad when\ \ type(\beta) \cap \mathfrak{D} = \emptyset \\ \bullet\ [\ \texttt{expand}([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\ \alpha_2]) \triangleright \texttt{expand}(\beta), \\ \quad [\texttt{expand}([\alpha_2]) \triangleright \texttt{expand}(\beta)]\ ]\quad otherwise \end{cases}$$

where $\mathfrak{R}' = \mathfrak{R} \cup \texttt{rigid}(\beta, X)$.

Fig. 15. Pushing antijoin in an equivalence node containing a fixpoint.

$$\text{pp}([\ \widetilde{\pi}_a\ \ [\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}]]) =$$

$$\begin{cases} \bullet\ [\ \mu X'.\ \texttt{expand}([\widetilde{\pi}_a(\gamma)]) \cup [\texttt{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}'}_{\mathfrak{D}'}\ ] \\ \quad when\ \ a \notin \mathfrak{R} \\ \bullet\ [\widetilde{\pi}_a(\texttt{expand}([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}]))\ ]\quad otherwise \end{cases}$$

$$\text{pp}([\ \widetilde{\pi}_a\ \ ([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}, \alpha_2])]) =$$

$$\begin{cases} \bullet\ [\ \mu X'.\ \texttt{expand}([\widetilde{\pi}_a(\gamma)]) \cup [\texttt{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}'}_{\mathfrak{D}'}\ , \\ \quad \texttt{expand}([\widetilde{\pi}_a([\alpha_2])])\ ]\quad when\ \ a \notin \mathfrak{R} \\ \bullet\ [\ \widetilde{\pi}_a(\texttt{expand}([\mu X.\ \gamma\ \cup\ [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}, \alpha_2]))\ , \\ \quad \texttt{expand}([\widetilde{\pi}_a([\alpha_2])])\ ]\quad otherwise \end{cases}$$

where $\mathfrak{D}' = \mathfrak{D} \cup \{a\}$ and $\mathfrak{R}' = \mathfrak{R} \cup \{a\}$.

Fig. 14. Pushing antiprojection in an equivalence node containing a fixpoint.

where `allCodd()` applies all rewrite rules concerning classical (non-recursive) relational algebra adapted for RLQDAG. For example: pfj() for pushing filters in join operation nodes, paj() for pushing antiprojections in a join operation node, jassoc() (for join associativity), dju() (for distributivity of join over union operation nodes) etc.

$$\begin{aligned} \texttt{allCodd}([d]) \quad = \quad & \text{pfj}([d]) \cup \text{paj}([d]) \cup \\ & \text{jassoc}([d]) \cup \ldots \cup \text{dju}([d]) \end{aligned}$$

For instance, pfj() is defined as shown in Fig. 16 for an RLQDAG in which a filter precedes an equivalence node which contains a join operation node. In other cases, pfj() recursively traverses the structure with appropriate calls to

$\text{pfj}(\lceil\sigma_f(\lceil\gamma_1 \bowtie \gamma_2, \alpha\rceil)\rceil) =$

$$\begin{cases} \bullet\ \lceil\ \texttt{exp}(\lceil\sigma_f(\gamma_1)\rceil) \bowtie \texttt{exp}(\gamma_2),\ \texttt{exp}(\lceil\sigma_f(\alpha)\rceil)\ \rceil \\ \quad \textit{when filt(f)} \subseteq \textit{type}(\gamma_1) \wedge \textit{filt(f)} \nsubseteq \textit{type}(\gamma_2) \\ \bullet\ \lceil\ \texttt{exp}(\gamma_1) \bowtie \texttt{exp}(\lceil\sigma_f(\gamma_2)\rceil),\ \texttt{exp}(\lceil\sigma_f(\alpha)\rceil)\ \rceil \\ \quad \textit{when filt(f)} \nsubseteq \textit{type}(\gamma_1) \wedge \textit{filt(f)} \subseteq \textit{type}(\gamma_2) \\ \bullet\ \lceil\ \texttt{exp}(\lceil\sigma_f(\gamma_1)\rceil) \bowtie \texttt{exp}(\lceil\sigma_f(\gamma_2)\rceil),\ \texttt{exp}(\lceil\sigma_f(\alpha)\rceil)\ \rceil \\ \quad \textit{when filt(f)} \subseteq \textit{type}(\gamma_1) \wedge \textit{filt(f)} \subseteq \textit{type}(\gamma_2) \\ \bullet\ \lceil\ \sigma_f(\texttt{exp}(\lceil\varphi \bowtie \psi\rceil),\ \texttt{exp}(\lceil\sigma_f(\alpha)\rceil))\ \rceil \quad \textit{otherwise} \end{cases}$$

Fig. 16. Pushing a filter in an equivalence node composed of at least one join operation node and other operation nodes (exp() stands for expand()).

expand() in search for further transformation opportunities. Other rewrite rules of non-recursive relational algebra are also implemented in a similar way in the RLQDAG.

C. Correctness and completeness

Proposition 1 (Correctness)**.** *Let* $\lceil\alpha\rceil$ *be a consistent RLQDAG, and* $\alpha' = \texttt{expand}(\lceil\alpha\rceil)$*, then we have* $S_\gamma[\![\lceil\alpha\rceil]\!]_\emptyset \subseteq S_\gamma[\![\alpha']\!]_\emptyset$ *and* α' *is consistent.*

Proof Sketch. The proof of correctness is done by induction on the structure of α', using a case-by-case analysis of syntactic decompositions (i.e. by considering separately each RLQDAG subcase and the corresponding case of the generalized rewrite rule that applies to it). The complete formal proof is too large to be listed here (with many straightforward subcases). We summarize the main principles as well as useful auxiliary properties. When proving correctness for each RLQDAG rewrite rule, (i) we focus on each newly created recursive term added by the expansion, and (ii) use the consistency hypothesis on the recursive term before transformation to apply the theorems for individual terms of [32] to each rearrangement of fixpoint operation node. We also need to show that the incremental updates of annotations performed by the RLQDAG rewrite rules preserve consistency. This is done for each update by leveraging the properties that both destab () and rigid() are distributive over union. □

Proposition 2 (Completeness properties)**.** *Let* R *be a set of RLQDAG rewrite rules such that* R *contains the 5 RLQDAG rewrite rules for recursive terms presented in Section IV, we consider* $\lceil\alpha\rceil = \texttt{unfold}(\texttt{expand}_R(\lceil\alpha'\rceil))$ *where* α' *is a consistent RLQDAG, and* $\texttt{expand}_R()$ *is the* expand() *function in which rules in* R *are activated. The following properties hold:*

Property 1 (No pushable filter left unpushed)**.**
$\forall\ \sigma_f(\gamma) \in \lceil\alpha\rceil,\ \nexists d \in \gamma \mid d = \mu X.\ [\kappa] \cup \lceil\alpha_2\rceil^{\mathfrak{R}}_{\mathfrak{D}}$ *and* $\text{filt(f)} \cap \mathfrak{D} = \emptyset$.

Property 2 (No pushable antiprojection left unpushed)**.**
$\forall\ \widetilde{\pi}_a(\gamma) \in \lceil\alpha\rceil,\ \nexists d \in \gamma \mid d = \mu X.\ [\kappa] \cup \lceil\alpha_2\rceil^{\mathfrak{R}}_{\mathfrak{D}}$ *and* $a \notin \mathfrak{R}$.

Property 3 (All pushable joins have been pushed)**.**
$\forall\ (\beta \bowtie \gamma) \in \lceil\alpha\rceil$*, if* $\exists d \in \gamma$ *such that* $d = \mu X.\ [\kappa] \cup \lceil\alpha_2\rceil^{\mathfrak{R}}_{\mathfrak{D}}$ *and* $\text{type}(\beta) \cap \mathfrak{D} = \emptyset$ *and* $\text{type}(\beta) \backslash \text{type}(\gamma) \cap \mathfrak{R} = \emptyset$ *then* $\exists d' \in \lceil\alpha\rceil$ *such that* $d' = \mu X'.\ [[\beta] \bowtie [\kappa]] \cup \lceil\alpha_{2\{X/X'\}}\rceil^{\mathfrak{R}'}_{\mathfrak{D}'}$.

Property 4 (All mergeable fixpoints have been merged)**.**
$\forall\ (\gamma_1 \bowtie \gamma_2) \in \lceil\alpha\rceil$*, if* $\mu X_1.\ [\kappa_1] \cup \lceil\alpha_1\rceil^{\mathfrak{R}_1}_{\mathfrak{D}_1} \in \gamma_1$ *and* $\mu X_2.\ [\kappa_2] \cup \lceil\alpha_2\rceil^{\mathfrak{R}_2}_{\mathfrak{D}_2} \in \gamma_2$ *and we have* $\gamma_1 \neq \gamma_2$ *and* $(\text{type}(\gamma_1) \cap \text{type}(\gamma_2)) \cap (\mathfrak{D}_1 \cup \mathfrak{D}_2) = \emptyset$ *and* $\text{type}(\gamma_1) \backslash \text{type}(\gamma_2) \cap \mathfrak{R}_2 = \emptyset$ *and* $\text{type}(\gamma_2) \backslash \text{type}(\gamma_1) \cap \mathfrak{R}_1 = \emptyset$ *then there exists* $d \in \lceil\alpha\rceil$ *such that* $d = \mu X.\ [[\kappa_1] \bowtie [\kappa_2]] \cup [[\alpha_1] \cup [\alpha_2]]^{\mathfrak{R}}_{\mathfrak{D}}$.

Property 5 (All pushable antijoins have been pushed)**.**
$\forall\ (\lceil\mu X.\ \gamma \cup [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}\rceil \triangleright \beta) \in \lceil\alpha\rceil$*, if* $\text{type}(\beta) \cap \mathfrak{D} = \emptyset$ *then* $\exists d \in \lceil\alpha\rceil$ *such that* $d = \lceil\mu X'.\ [\gamma \triangleright \beta] \cup [\alpha_{\{X/X'\}}]^{\mathfrak{R}'}_{\mathfrak{D}}\rceil$

Proof Sketch. These properties are proved by contradiction: (i) assuming the existence of a missed transformation opportunity in the expansion, which (ii) necessarily implies some unrealized rule application (whereas the rule was applicable), and (iii) showing that the systematic structure traversal performed by $\texttt{expand}_R()$ leaves no room for such a missed opportunity, thus (iv) leading to a contradiction.

□

V. Experimental Results

We evaluate the RLQDAG experimentally. Our assessment is driven by the following research questions:

RQ1 How efficient is RLQDAG exploration of recursive plan spaces compared to the state-of-the-art?

RQ2 How relevant are explorations of large recursive plan spaces for practical query evaluation?

In the following subsections we first introduce the experimental setup and the experimental methodology with chosen baselines and metrics. We then report on the results and the main lessons learned.

A. Experimental setup

a) Datasets: We consider various datasets, graphs and trees, real and generated, as described in Table I.

Dataset	#nodes & #edges	Type	Nature
Yago [62]	42M & 62M	Knowledge graph	Real
Bahamas Leaks [8]	202K & 249K	Property graph	Real
Airbnb [6]	24K & 14K	Property graph	Real
LDBC [11]	908K & 1.9M	Property graph	Synthetic
Wikitree [19]	9.1M & 1.3M	Tree	Real
Academic tree [36]	765K & 1.5M	Tree	Real

TABLE I
DATASETS (AVAILABLE FROM [54]).

b) Queries: We consider a variety of recursive queries formulated against these datasets. Queries for Yago are mainly third-party regular path queries already considered in earlier papers in the literature[5], and chosen because they are representative of the variety of possible recursive optimizations that can apply to them. Queries over the Airbnb dataset are inspired from [48]. We added more queries formulated over the Bahamas and LDBC datasets. We consider non-regular queries (variants of same generation, and $a^n b^n$) for the Wikitree

[5]We consider 7 queries (Q1-Q7) taken from [3], 2 queries (Q8-Q9) taken from [63], (Q10-Q11) taken from [28] and (Q12-Q20) come from [32].

and Academic Tree datasets. Queries and datasets used in experiments are available at [54].

c) Hardware: All experiments are conducted on a machine with an Intel Xeon 2.20GHz CPU and 192Gb of RAM.

B. ***RQ1**: Efficiency of plan space exploration.*

To answer **RQ1**, we implemented a prototype of the RLQDAG and compare it with `MuEnum`, which is the plan enumerator of the state-of-the-art μ-RA system [32]. As described in Section II, this system is of the most advanced relational-based system for recursive query optimization; providing the richest plan spaces for recursive terms. `MuEnum` [32] explores them using a state-of-the-art dynamic programming strategy, in which terms are made unique in memory so as to obtain very efficient term equality tests.

We measure the enumeration capability of the RLQDAG in terms of the number of plans explored per second for each query. We compare the performance of our RLQDAG prototype implementation with the performance of `MuEnum`. Figures 17, 18, 19, and Fig. 20 respectively show the results obtained for the queries over each dataset.

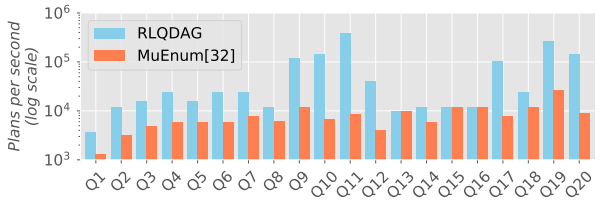

Fig. 17. Yago queries.

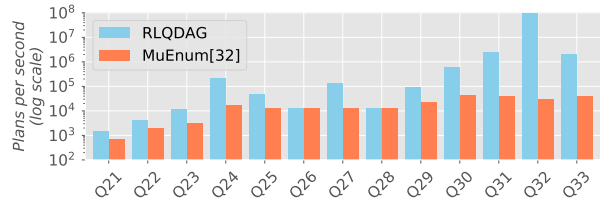

Fig. 18. Bahamas Leaks queries.

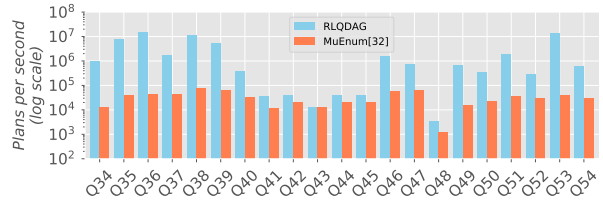

Fig. 19. Airbnb and LDBC queries.

In these figures, the y axis (in log scale) indicates the number of plans per second explored by each approach for a given query (on the x axis). Results suggest that the RLQDAG approach always enumerates plans much faster (up to two orders of magnitude) when compared to `MuEnum`[6].

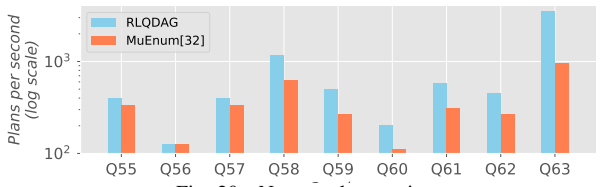

Fig. 20. Non-regular queries.

Now, we set a time budget t (in seconds) for the plan space exploration and let the two systems generate plan spaces for that time budget. This means that after t elapsed seconds we stop the two explorations and look at the plan spaces obtained by the two systems. Fig. 21 shows the results obtained for a time budget of $t = 0.5$ seconds with queries from Bahamas Leaks (Q31-Q32), Airbnb (Q34-Q37-Q38-Q39) and LDBC (Q51-Q53-Q54). The y axis (in log scale) indicates the number of plans found. Fig. 21 also indicates the size of the complete (exhaustive) plan space obtained without any time restriction for the plan exploration ($t = \infty$). For example, for query Q31, the complete plan space contains more than 21.4 million plans. In 0.5 seconds, the RLQDAG prototype explored 1,019,026 plans whereas `MuEnum` explored only 5,751 plans. This is because although `MuEnum` uses dynamic programming techniques, it is not capable of benefitting from the RLQDAG's grouping effect when applying complex rewrite rules on sets of recursive terms at once, thus rules are significantly more costly to apply. Seen from another perspective, this means that the RLQDAG's approach is more effective in avoiding redundant subcomputations. We have conducted extensive experiments and overall results indicate that, for a given time budget, the RLQDAG prototype explores many more terms in comparison to `MuEnum`, in all cases. In some cases, the RLQDAG generates a number of plans which is greater by up to two orders of magnitude (for the same time budget). Such a speedup sometimes enables a complete exploration of the whole plan space in some cases (as shown e.g. for Q34, Q51, Q53, Q54 in Fig. 21).

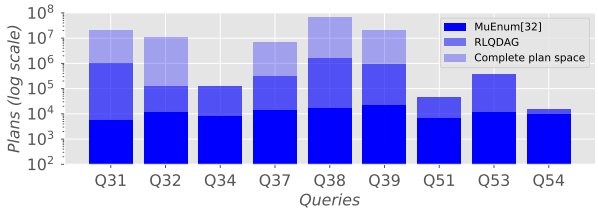

Fig. 21. Plan spaces explored in 500ms.

We now report on experiments of exploring plan spaces with varying and increasing time budgets for the same query. For instance, Fig. 22 presents the number of plans explored (on the y axis) for different time budgets shown on the x axis for query Q31 and query Q53 (2 queries from 2 different datasets).

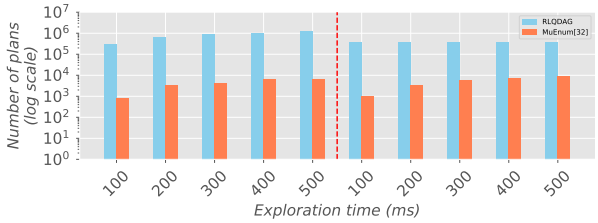

Fig. 22. Plans explored per time budgets for Q31 and Q53.

[6]Cases where histogram bars look very similar correspond to situations where both systems completed the plan space exploration in a very short time duration, which makes the difference hardly visible with the log scale.

Again, results shown in Fig. 22 indicate that the RLQDAG explores significantly more plans than the other approach for all considered time budgets (the y axis is in log scale). We can also observe that the difference between the amount of plans explored by each system stays of the same order, even when exploration time increases.

C. *RQ2: Relevance of exploring large plan spaces.*

To answer **RQ2**, we use the same backend (PostgreSQL) in order to execute query plans generated by `MuEnum` and RLQDAG. For a given query, we set a time budget for plan space exploration, and we let `MuEnum` and RLQDAG generate plan spaces for this same time budget. Now, we pick the best estimated plan from each generated plan space. We use the same heuristics for cost estimations [35], thus making relevant a head to head comparison. We measure and compare the times spent by PostgreSQL for evaluating the plans.

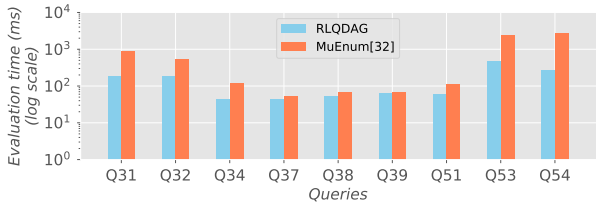

Fig. 23. Evaluating best estimated plans found in explored spaces of Fig 21.

For example, Fig. 23 illustrates the time spent in evaluating the best estimated plan taken from each of the explored plan spaces reported in Fig. 21. Results show the benefits of exploring a much larger plan space: the RLQDAG approach always provides similar or better performance, which is a direct consequence of the availability of more efficient recursive plans in the larger plan space explored.

Furthermore, for the same query, we also measure the time spent in evaluating the best estimated plans from the different plan spaces explored with a varying time budget. For instance, Fig. 24 shows the time spent in evaluating the best plans selected from each plan space explored with the time budget shown on the x axis. The sizes of the corresponding plan spaces are given in Fig. 22. Results shown in Fig. 24 show that the best estimated term selected from the larger plan space is more efficiently evaluated. This suggests that in practice, larger plan spaces are very prone to contain more efficient recursive plans. This confirms the interest of efficiently exploring large recursive plan spaces.

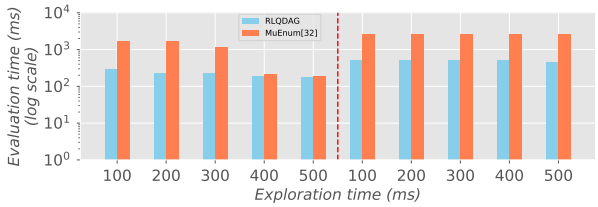

Fig. 24. Evaluating best estimated plans found in explored spaces of Fig 22.

Finally, we assess to which extent the availability of more relational algebraic plans can be useful in practice, when compared to other RDBMS and state-of-the-art approaches in graph query evaluation. For this purpose, we consider 2 engines based on relational algebra (`MuEnum` [32] and Virtuoso [15] version 7.2.6.1), 3 native graph database engines (MilleniumDB [56], Neo4j [60] version 4.4.11, Blazegraph [53] version 2.1.6), and 3 plain-vanilla RDBMS capable of evaluating recursive SQL queries (PostgreSQL [50] version 15.1, MySQL [1] version 8.1.0, SQLite [30] version 3.36.0). We measure the time spent in query evaluation with each system. For each system, this includes the time spent in query optimization and the time spent in retrieving the whole set of query results. We set a timeout of 600s (10 min). Fig. 25 shows the corresponding times spent for systems which were able to answer. If a system does not answer before the timeout, we consider that it is unable to answer, and it is absent from Fig. 25 for the given query.

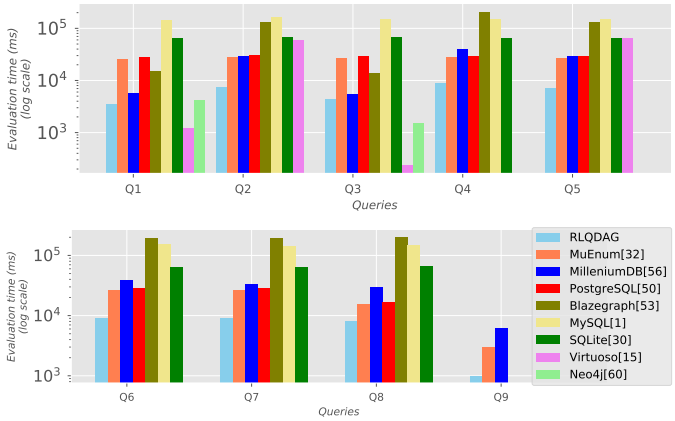

Fig. 25. Comparative evaluation of third-party queries (excerpt from [54]).

Results suggest that the availability of more plans – thanks to the RLQDAG – is beneficial to the relational-based approach. More specifically, existing RDBMS consider recursive queries in a very restricted way. Most of them optimize recursion-free subexpressions, without being able to optimize the recursive part as a whole: they cannot make significant structural changes to the entire recursive query. This has an important consequence: they do explore much smaller plan spaces when compared to RLQDAG, potentially missing very efficient plans. In some cases, it even makes the relational-based approach more efficient than specialized graph engines.

VI. Conclusion and Perspectives

We propose the RLQDAG for capturing and efficiently transforming sets of recursive relational terms. This is done by introducing annotated equivalence nodes, and a formal syntax and semantics for RLQDAG terms that enable the development of RLQDAG rewrite rules on a solid ground. RLQDAG rewrite rules transform sets of recursive terms while precisely describing how new subterms are created, attached, shared, and how new structural annotations are obtained with incremental updates. Practical experiments with the RLQDAG show the interest of exploring large plan spaces, and suggest that it represents an interesting foundation for efficiently enumerating recursive relational query plans.

REFERENCES

[1] MySQL, 2023. https://www.mysql.com/.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995.

[3] Z. Abul-Basher, N. Yakovets, P. Godfrey, S. Ghajar-Khosravi, and M. H. Chignell. Tasweet: optimizing disjunctive regular path queries in graph databases. In *EDBT/ICDT 2017 joint conference 20th international conference on extending database technology. https://doi.org/10.5441/002/edbt*, 2017.

[4] 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.

[5] A. V. Aho and J. 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.

[6] Airbnb. Airbnb. http://insideairbnb.com/get-the-data.html, 2022.

[7] P. Alvaro, W. Marczak, N. Conway, J. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. pages 262–281, 01 2010.

[8] Bahamas-Leaks. Bahamas leaks. https://offshoreleaks.icij.org/pages/about, 2016.

[9] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, page 1–15, New York, NY, USA, 1985. Association for Computing Machinery.

[10] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, page 16–52, New York, NY, USA, 1986. Association for Computing Machinery.

[11] P. Boncz. LDBC: Benchmarks for graph and RDF data management. In *Proceedings of the 17th International Database Engineering ; Applications Symposium*, IDEAS '13, page 1–2, New York, NY, USA, 2013. Association for Computing Machinery.

[12] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, page 34–43, New York, NY, USA, 1998. Association for Computing Machinery.

[13] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, volume 94, pages 12–15, 1994.

[14] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[15] O. Erling. Virtuoso, a hybrid RDBMS/graph column store. *IEEE Data Eng. Bull.*, 35:3–8, 2012.

[16] Z. Fan, J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, and J. M. Patel. Scaling-up in-memory Datalog processing: Observations and techniques. *Proc. VLDB Endow.*, 12(6):695–708, 2019.

[17] P. Fender and G. Moerkotte. Counter strike: Generic top-down join enumeration for hypergraphs. *Proc. VLDB Endow.*, 6(14):1822–1833, sep 2013.

[18] P. Fender and G. Moerkotte. Top down plan generation: From theory to practice. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1105–1116, 2013.

[19] M. Fire and Y. Elovici. Data mining of online genealogy datasets for revealing lifespan patterns in human population. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6(2):28, 2015.

[20] M. Francis-Landau, T. Vieira, and J. Eisner. Evaluation of logic programs with built-ins and aggregation: A calculus for bag relations, 10 2020.

[21] C. Galindo-Legaria and A. Rosenthal. How to extend a conventional optimizer to handle one-and two-sided outerjoin. In *1992 Eighth International Conference on Data Engineering*, pages 402–403. IEEE Computer Society, 1992.

[22] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems (TODS)*, 22(1):43–74, 1997.

[23] G. Gardarin. Magic functions: A technique to optimize extended Datalog recursive programs. pages 21–30, 01 1987.

[24] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. Technical report, Stanford, 1997.

[25] G. Graefe. The Cascades framework for query optimization. *Data Engineering Bulletin*, 18, 1995.

[26] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 160–172, New York, NY, USA, 1987. ACM.

[27] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society.

[28] A. Gubichev, S. J. Bedathur, and S. Seufert. Sparqling kleene: fast property paths in RDF-3X. In *First International Workshop on Graph Data Management Experiences and Systems*, pages 1–7, 2013.

[29] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, page 377–388, New York, NY, USA, 1989. Association for Computing Machinery.

[30] R. D. Hipp. SQLite, 2023. https://www.sqlite.org.

[31] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: An interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 1213–1216, New York, NY, USA, 2011. Association for Computing Machinery.

[32] L. Jachiet, P. Genevès, N. Gesbert, and N. Layaïda. On the optimization of recursive relational queries: Application to graph queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 681–697, 2020.

[33] M. Kifer and E. L. Lozinskii. On compile-time query optimization in deductive databases by means of static filtering. *ACM Trans. Database Syst.*, 15(3):385–426, sep 1990.

[34] A. Koschmieder and U. Leser. Regular path queries on large graphs. In *International Conference on Scientific and Statistical Database Management*, pages 177–194. Springer, 2012.

[35] M. Lawal, P. Genevès, and N. Layaïda. A cost estimation technique for recursive relational algebra. *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020.

[36] J. F. Liénard, T. Achakulvisut, D. E. Acuna, and S. V. David. Intellectual synthesis in mentorship determines success in academic careers. *Nature communications*, 9(1):1–13, 2018.

[37] G. M. Lohman. Grammar-like functional rules for representing query optimization alternatives. *SIGMOD Rec.*, 17(3):18–27, June 1988.

[38] G. M. Lohman, C. Mohan, L. M. Haas, D. Daniels, B. G. Lindsay, P. G. Selinger, and P. F. Wilms. *Query Processing in R**, pages 31–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985.

[39] G. Moerkotte, P. Fender, and M. Eich. On the correct and complete enumeration of the core search space. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 493–504, 2013.

[40] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. Efficient evaluation of right-, left-, and multi-linear rules. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, page 235–242, New York, NY, USA, 1989. Association for Computing Machinery.

[41] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. *SIGMOD Rec.*, 29(2):249–260, May 2000.

[42] D. Saccà and C. Zaniolo. On the implementation of a simple class of logic queries for databases. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, page 16–23, New York, NY, USA, 1985. Association for Computing Machinery.

[43] M. Schleich, A. Shaikhha, and D. Suciu. Optimizing tensor programs on flexible storage. *Proc. ACM Manag. Data*, 1(1):37:1–37:27, 2023.

[44] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[45] J. Seo, S. Guo, and M. S. Lam. SociaLite: An efficient graph query language based on Datalog. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1824–1837, 2015.

[46] A. Shanbhag and S. Sudarshan. Optimizing join enumeration in transformation-based query optimizers. *Proc. VLDB Endow.*, 7(12):1243–1254, aug 2014.

[47] L. Shapiro, D. Maier, P. Benninghoff, K. Billings, Y. Fan, K. Hatwal, Q. Wang, Y. Zhang, H.-M. Wu, and B. Vance. Exploiting upper and lower bounds in top-down query optimization. pages 20 – 33, 02 2001.

[48] C. Sharma, R. Sinha, and K. Johnson. Practical and comprehensive formalisms for modelling contemporary graph query languages. *Information Systems*, 102:101816, 2021.

[49] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1135–1149, New York, NY, USA, 2016. Association for Computing Machinery.

[50] M. Stonebraker and L. A. Rowe. The design of postgres. SIGMOD '86, page 340–355, New York, NY, USA, 1986. Association for Computing Machinery.

[51] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. *Log. Methods Comput. Sci.*, 7(1), 2011.

[52] K. T. Tekle and Y. A. Liu. More efficient Datalog queries: Subsumptive tabling beats magic sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 661–672, New York, NY, USA, 2011. Association for Computing Machinery.

[53] B. Thompson, M. Personick, and M. Cutcher. The bigdata® RDF graph database. In *Linked Data Management*, pages 221–266. Chapman and Hall/CRC, 2016.

[54] Tyrex-repository. Datasets and queries used in experiments with the RLQDAG. https://gitlab.inria.fr/tyrex-public/rlqdag, 2023.

[55] V. Vianu. Datalog unchained. In L. Libkin, R. Pichler, and P. Guagliardo, editors, *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021*, pages 57–69. ACM, 2021.

[56] D. Vrgoč, C. Rojas, R. Angles, M. Arenas, D. Arroyuelo, C. Buil-Aranda, A. Hogan, G. Navarro, C. Riveros, and J. Romero. Millenniumdb: An open-source graph database system. *Data Intelligence*, pages 1–39, 2022.

[57] J. Wang, M. Balazinska, and D. Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *Proc. VLDB Endow.*, 8(12):1542–1553, aug 2015.

[58] Y. R. Wang, S. Hutchison, D. Suciu, B. Howe, and J. Leang. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13(11):1919–1932, 2020.

[59] Y. R. Wang, M. A. Khamis, H. Q. Ngo, R. Pichler, and D. Suciu. Optimizing recursive queries with program synthesis. *arXiv preprint arXiv:2202.10390*, 2022.

[60] J. Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 217–218, New York, NY, USA, 2012. Association for Computing Machinery.

[61] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.

[62] YAGO. Yago: A high-quality knowledge base. https://www.mpi-inf.mpg.de/yago-naga/yago/, july 2019.

[63] N. Yakovets, P. Godfrey, and J. Gryz. WAVEGUIDE: Evaluating SPARQL property path queries. In *EDBT*, volume 2015, pages 525–528, 2015.

[64] H. Zauner, B. Linse, T. Furche, and F. Bry. A RPL through RDF: Expressive navigation in RDF graphs. In *International Conference on Web Reasoning and Rule Systems*, pages 251–257. Springer, 2010.