Efficient Enumeration of Recursive Plans in Transformation-based Query Optimizers

Amela Fejza
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France
amela.fejza@inria.fr

Pierre Genevès
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France
pierre.geneves@inria.fr

Nabil Layaïda
Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
Grenoble, France
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, are restricted to recursion-free queries. 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, enabling 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.

PVLDB Reference Format:
Amela Fejza, Pierre Genevès, and Nabil Layaïda. Efficient Enumeration of Recursive Plans in Transformation-based Query Optimizers. PVLDB, 17(11): 3095 - 3108, 2024.
doi:10.14778/3681954.3681986

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://gitlab.inria.fr/tyrex-public/rlqdag.

1 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 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 well-studied for recursion-free queries. Allocating a time budget for this enumeration phase is common, 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 more efficient plans. 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. Plan enumeration speed directly determines whether query evaluation plans theoretically enabled by e.g. μ-RA [32] are within range of a practical query optimizer or not. This motivates the search for efficient recursive plan enumeration methods, as they unlock potential for big performance gains, and are possibly decisive for the feasibility of certain queries.

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 (i) the first extension of the LQDAG with the support of recursive terms; (ii) 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; (iii) RLQDAG transformations with incremental annotation updates. These transformations generalize rewrite rules from individual recursive terms (such as those of [32]) to *grouped transformations of compactly represented sets of recursive terms*. This enables much more efficient explorations of recursive plan spaces, which in turn makes available in practice very efficient evaluation plans unmatched by previous techniques. For this to be possible, the RLQDAG relies on a concept of annotated equivalence nodes with incremental updates, used for guiding transformations of recursive subterms. Contributions also include (iv) a complete implementation of the proposed approach and experimental assessments using third-party queries on synthetic and real datasets.

2 BACKGROUND AND RELATED WORK

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.

Proceedings of the VLDB Endowment, Vol. 17, No. 11 ISSN 2150-8097.
doi:10.14778/3681954.3681986

The Datalog line of works [7, 16, 20, 31, 47, 51, 57, 59] developed methods for optimizing recursive queries formulated in Datalog: magic-sets [9, 23, 44], demand transformations [54], automated reversals [42], and the FGH rule [61]. Although the syntax of Datalog greatly differs from RA, the effects of magic-sets [9, 23, 44] and of demand transformations [54] 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 [42] to fully exploit them. The framework proposed in [61] 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, 34] 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 [34]. 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 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 4) apply directly on a factorized representation of the recursive plan space. Among the benefits, this enables (1) applying transformations on sets of algebraic terms at once (instead of successively on individual terms), and (2) exploiting the sharing of common subterms to avoid redundant computations.

Ad-hoc optimizations for regular queries. Both Datalog 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, 35, 66]. A hybrid approach that combines finite state machines and α-RA is presented in [65] 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 5 with queries of the form A^nB^n for instance).

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.

2.1 Plan enumeration for non-recursive 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 [29, 39, 40, 45, 46, 53, 60, 63] 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 [25–27] start from the root and recursively explore subbranches in search for possible alternatives. An advantage of the top-down approach is that it enables branch and bound pruning [49].

All the previous approaches focus mainly on SPJ queries, with some extensions to support outer joins [17, 18, 21, 22, 41] and aggregations [13]. Very few works consider other operators, as noticed in [12] and [48]. 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.

2.2 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]. It is also well described as the AND-OR-DAG in [43, 48] where it is used for detecting and unifying common subexpressions for multi-query optimization [43]; and for generating the space of cross-product free join trees [48].

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

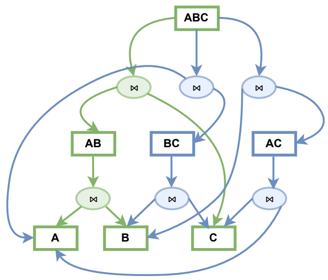

Figure 1: Sample LQDAG (in green) with expansion (in blue).

3 THE RLQDAG

The RLQDAG introduces a novel representation designed to effectively capture and transform sets of recursive terms. This representation enables the delineation of subclasses within recursive terms, based on shared properties. This segmentation facilitates the grouping of similar terms, streamlining the process of collectively transforming them in a single operation.

3.1 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
		$[\alpha]$	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}	::=	$[\alpha]^{\mathfrak{R}}_{\mathfrak{D}}$	Annotated equivalence node

Figure 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 $[\alpha]$ 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 $\rho_{\mathrm{a}}^{\mathrm{b}}(\gamma)$ 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] and generalized here to sets of terms, consists in the introduction of a least-fixpoint binder operation node (μ) that binds a fresh variable X to some expression in which X can appear, thus explicitly denoting recursion. Our generalization is defined in the syntax of Fig. 2 and illustrated in Fig. 3. A fixpoint operator node is written $\mu X.\ \gamma \ \cup \ \alpha_{rec}$. The operand γ is an equivalence node that models the constant part (the base case) of the recursion. X cannot occur within γ. The equivalence node α_{rec} is the recursive part. An essential consideration is that the α_{rec} branch contains at least one free occurrence of the recursive variable X. This characteristic distinguishes the fixpoint operation node from other operation nodes. It will lead to a number of new definitions and formal developments. Intuitively, this is because depending on how the recursive variable is used in that branch, transformation and sharing of RLQDAG subterms may, or may not, be allowed.

For example, on the Yago graph dataset [64], the query 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 Q_{e_1}:

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

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 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.

The recursive equivalence node of each fixpoint operation node is annotated with $\mathfrak{D}$ and $\mathfrak{R}$. These annotations will be key for guiding the application of transformations (see Section 3.4).

There are two reasons why we distinguish α_{rec} from a general equivalence node α in the abstract syntax. The first reason is that equivalence nodes for recursive parts are equipped with annotations (see Section 3.5.). 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.

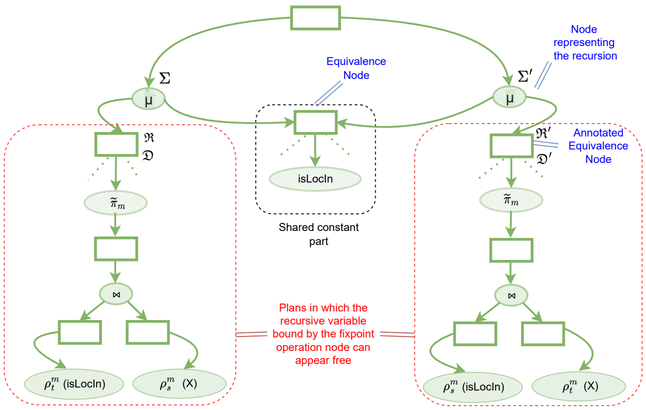

Figure 3: Structure of recursive terms in RLQDAG.

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

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

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 expressive queries containing union, conjunction, transitive closure of arbitrary expressions and non-regular patterns such as $A^n B^n$.

3.2 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, and $E \oplus \{Y \mapsto \alpha_1\}$ denotes the environment E in which variable Y is bound to α_1. The interpretation of a RLQDAG $[\alpha]$ is $S_{\alpha}[\![\alpha]\!]_{\emptyset}$.

A well-formed RLQDAG is a RLQDAG whose interpretation is a set of semantically equivalent terms. For example, Fig. 5 illustrates a well-formed RLQDAG capturing two semantically equivalent relational terms obtained before and after 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.

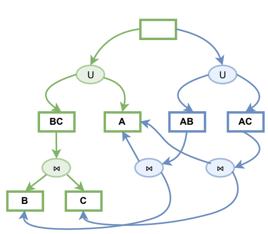

Figure 5: Well-formed.

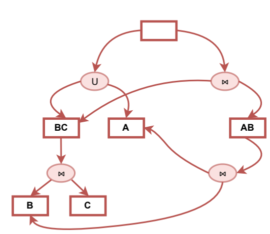

Figure 6: Not well-formed.

The following generalizes well-formedness to encompass recursion:

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 set-semantics interpretation of the individual recursive relational algebraic term t (i.e. the set of tuples returned by t when evaluated in a database instance).

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.

3.3 Recursive terms and rule applicability

A significant novelty introduced by recursion resides in the criteria used to trigger rewrite rules. In the 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 traversals 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 complex 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, 34] that criteria for rule application are significantly more complex in the presence of recursion as they need to examine how the recursive variables are used. A key contribution of this work is to show that it is still possible to apply rules over sets of recursive terms at once, using the new concept of *annotated equivalence nodes* (introduced in § 3.5). We first need some preliminary definitions.

3.4 Preliminary definitions for 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 the new concept of annotated equivalence nodes introduced in Section 3.5.

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, 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: $\texttt{destab}(\alpha', X) = \{c \in \mathfrak{C} \mid \exists p \in \texttt{d}(\alpha', X)\ p(c) \neq c\}$ where $\mathfrak{C}$ is an infinite set of column names and $\texttt{d}(\cdot,\cdot)$ computes the set of derivations over a RLQDAG term:*

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

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.

Notion of rigidity in RLQDAG. We define a function 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).

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

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.

3.5 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: $[\alpha]_{\mathfrak{D}}^{\mathfrak{R}}$ where $\mathfrak{D} = \texttt{destab}(\alpha, X)$ and $\mathfrak{R} = \texttt{rigid}(\alpha, X)$.*

For example, in Fig. 3's RLQDAG, 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'. They will be used for guiding transformations of recursive RLQDAG terms. For instance, on Σ, pushing a filter on t is possible but not on s. A pair (s, t) might not pass the filter but still be useful to discover a query answer (s', t) that passes the filter.

Annotations are intended to characterize all the subterms of the annotated equivalence node (thanks to definition 6). Annotated equivalence nodes constitute a novel notion whose goal 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.

Notion of consistency. Intuitively, a RLQDAG is *consistent* iff it is well-formed and in addition, for any annotated equivalence node $[\alpha_2]_{\mathfrak{D}}^{\mathfrak{R}}$, 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 $[\alpha]$ is consistent iff:*

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

$$
\left\{
\begin{array}{rcl}
\texttt{cons}([d],X)_{\mathfrak{D}}^{\mathfrak{R}} & \stackrel{def}{=} & \texttt{destab}(d,X)=\mathfrak{D}\ \wedge\ \texttt{rigid}(d,X)=\mathfrak{R} \\
\texttt{cons}([d,\alpha],X)_{\mathfrak{D}}^{\mathfrak{R}} & \stackrel{def}{=} & \texttt{destab}(d,X)=\mathfrak{D}\ \wedge\ \texttt{rigid}(d,X)=\mathfrak{R} \\
 & & \wedge\ \texttt{cons}([\alpha],X)_{\mathfrak{D}}^{\mathfrak{R}}
\end{array}
\right.
$$

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.

4 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.

4.1 RLQDAG rewrite rules

We formalize all these ideas by introducing RLQDAG rewrite rules, based on the syntax of RLQDAG terms introduced in Section 3. Specifically, RLQDAG rewrite rules are formalized as functions that take an equivalence node γ and return another equivalence node γ' obtained after applying transformations.

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 $\mathrm{pf}(\big[d\big])$ and $\mathrm{pf}(\big[d, \alpha\big])$ 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].

In Fig. 7, whenever the filter can be pushed through the fixpoint operation node, pf() generates a new RLQDAG subterm "*pushed*" in which the filter operation node is put within the constant part of the fixpoint operation node.

For this transformation to happen, the criteria *filt(f)* $\cap\, \mathfrak{D}$ must be satisfied. Whenever it is not the case, the filter is not rearranged but the RLQDAG structure is simply recursively traversed in search for more transformation opportunities. 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 4.2. 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. Notice that the function pf() takes a parameter *rep* as input. This parameter is used to control whether the initial term (*unpushed*) is preserved or not in the expansion. For instance, pf(·, false) will keep the initial term whereas pf(·, true) will replace it by the term "*pushed*" in which the filter is pushed. When the parameter *rep* is omitted, it is assumed to be false.

For example, the query Q_{e_2}:

$$?s \longleftarrow ?s\ \texttt{isLocatedIn+ Sweden}$$

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 [64]. Fig. 8 illustrates graphically a portion of 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 $\mathfrak{R}' = \mathfrak{R} \cup$ *filt(f)*.

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 opportunities 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 Q_{e_3}:

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

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 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.

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

$$\begin{array}{lcl} \mathrm{pf}(\mu X.\, \gamma \cup \big[\alpha\big]^{\mathfrak{R}}_{\mathfrak{D}}) & = & \mu X.\, \mathsf{expand}(\gamma) \cup \big[\mathsf{expand}(\alpha)\big]^{\mathfrak{R}}_{\mathfrak{D}} \\ \mathrm{pf}(\big[\big[A\big] \bowtie \big[B\big]\big]) & = & \mathsf{expand}(\big[A\big]) \bowtie \mathsf{expand}(\big[B\big]) \end{array}$$

where the expand() function, defined in Section 4.2, 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.

$$\mathrm{pf}([\sigma_f([\mu X.\,\gamma \cup [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}])], rep) =$$
$$\begin{cases} \bullet \; [\,\textit{unpushed}, \textit{pushed}\,] \text{ when } \textit{filt}(f) \cap \mathfrak{D} = \emptyset \text{ and } rep = \text{false} \\ \bullet \; [\,\textit{pushed}\,] \text{ when } \textit{filt}(f) \cap \mathfrak{D} = \emptyset \text{ and } rep = \text{true} \\ \bullet \; [\,\textit{unpushed}\,] \text{ otherwise} \end{cases}$$

$$\mathrm{pf}([\sigma_f([\mu X.\,\gamma \cup [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\, \alpha_2])], rep) =$$
$$\begin{cases} \bullet \; [\,\textit{unpushed}_2, \textit{pushed}, \textit{expand}_{\alpha_2}\,] \text{ when } \textit{filt}(f) \cap \mathfrak{D} = \emptyset \text{ and } rep = \text{false} \\ \bullet \; [\,\textit{pushed}, \textit{expand}_{\alpha_2}\,] \text{ when } \textit{filt}(f) \cap \mathfrak{D} = \emptyset \text{ and } rep = \text{true} \\ \bullet \; [\,\textit{unpushed}_2, \textit{expand}_{\alpha_2}\,] \text{ otherwise} \end{cases}$$

where:

$$\begin{aligned} \textit{pushed} &= \mu X'.\, \mathrm{expand}([\sigma_f(\gamma)]) \cup [\mathrm{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}}_{\mathfrak{D}} \\ \textit{unpushed} &= \sigma_f(\mathrm{expand}([\mu X.\,\gamma \cup [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}])) \\ \textit{unpushed}_2 &= \sigma_f(\mathrm{expand}([\mu X.\,\gamma \cup [\alpha]^{\mathfrak{R}}_{\mathfrak{D}},\, \alpha_2])) \\ \textit{expand}_{\alpha_2} &= \mathrm{expand}([\sigma_f([\alpha_2])]) \\ \alpha_{\{X/X'\}} &= \alpha \text{ in which all occurrences of } X \text{ are replaced by } X'. \end{aligned}$$

Figure 7: Pushing a filter in an equivalence node containing a fixpoint.

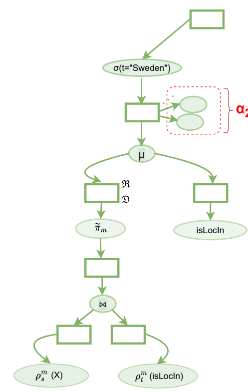

Figure 8: Initial RLQDAG before expansion with opportunity to apply pf().

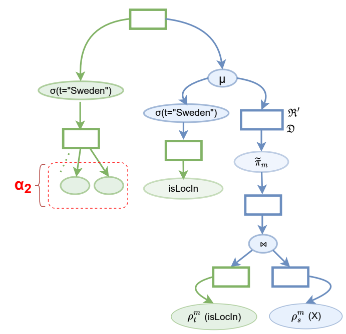

Figure 9: RLQDAG expansion by pushing a filter in a fixpoint operation node, with branches added by pf(·, true) in blue.

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

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

Figure 10: Pushing join in an equivalence node containing a fixpoint.

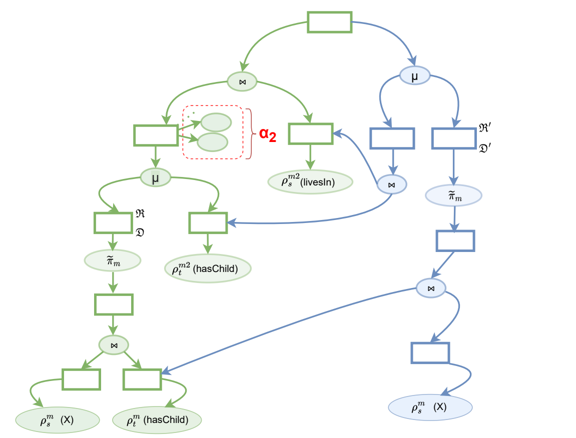

Figure 11: Expansion of a RLQDAG by pushing a join in a fixpoint operation node. The initial RLQDAG is in green color, and parts added by pj() are in blue color.

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 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 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 Q_{e_4} RLQDAG, with the new recursive term produced, using subterm-sharing, after the merging of fixpoint operation nodes.

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 are satisfied, and this results in the creation of a new term in the equivalence node. Depending on the value of the parameter *rep* (false when omitted), the initial term is either preserved or discarded in the expansion. Whenever the criteria is not satisfied, the initial term is left unchanged, but traversed in search for more transformation opportunities.

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

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

$$\mathrm{mf}(\,[\,[\mu X_1.\,\gamma_1 \cup [\alpha_1]^{\mathfrak{R}_6}_{\mathfrak{D}_6}] \bowtie [\mu X_2.\,\gamma_2 \cup [\alpha_2]^{\mathfrak{R}_7}_{\mathfrak{D}_7}]\,]\,) =$$

$$\begin{cases} \bullet & [\text{ let } \mathrm{const}_1 = \gamma_1 \text{ in, let } \mathrm{const}_2 = \gamma_2 \text{ in} \\ & \quad \mathtt{expand}([\mu X_1.\,\mathrm{const}_1 \cup [\alpha_1]^{\mathfrak{R}_6}_{\mathfrak{D}_6}]) \bowtie \mathtt{expand}([\mu X_2.\,\mathrm{const}_2 \cup [\alpha_2]^{\mathfrak{R}_7}_{\mathfrak{D}_7}])\,, \\ & \quad \mu X.\,\mathtt{expand}([\mathrm{const}_1 \bowtie \mathrm{const}_2]) \cup \mathtt{expand}([\alpha_{1\{X_1/X\}} \,\cup\, \alpha_{2\{X_2/X\}}]^{\mathfrak{R}}_{\mathfrak{D}})\,] \\ & \text{when } (type(\gamma_1) \cap type(\gamma_2)) \cap (\mathfrak{D}_6 \cup \mathfrak{D}_7) = \emptyset \\ & \text{and } type(\gamma_1)\backslash type(\gamma_2) \cap \mathfrak{R}_7 = \emptyset \text{ and } type(\gamma_2)\backslash type(\gamma_1) \cap \mathfrak{R}_6 = \emptyset \\ \bullet & [\,\mathtt{expand}([\mu X_1.\,\mathrm{const}_1 \cup [\alpha_1]^{\mathfrak{R}_6}_{\mathfrak{D}_6}]) \bowtie \mathtt{expand}([\mu X_2.\,\mathrm{const}_2 \cup [\alpha_2]^{\mathfrak{R}_7}_{\mathfrak{D}_7}])] \\ & \text{otherwise} \end{cases}$$

$$\mathrm{mf}(\,[\,[\mu X_1.\,\gamma_1 \cup [\alpha_1]^{\mathfrak{R}_6}_{\mathfrak{D}_6}, \alpha_3] \bowtie [\mu X_2.\,\gamma_2 \cup [\alpha_2]^{\mathfrak{R}_7}_{\mathfrak{D}_7}, \alpha_4]\,]\,) =$$

$$\begin{cases} \bullet & [\text{ let } \mathrm{const}_1 = \gamma_1 \text{ in, let } \mathrm{const}_2 = \gamma_2 \text{ in} \\ & \quad \mathtt{expand}([\mu X_1.\,\mathrm{const}_1 \cup [\alpha_1]^{\mathfrak{R}_6}_{\mathfrak{D}_6}]) \bowtie \mathtt{expand}([\mu X_2.\,\mathrm{const}_2 \cup [\alpha_2]^{\mathfrak{R}_7}_{\mathfrak{D}_7}])\,, \\ & \quad \mu X.\,\mathtt{expand}([\mathrm{const}_1 \bowtie \mathrm{const}_2]) \cup \mathtt{expand}([\alpha_{1\{X_1/X\}} \,\cup\, \alpha_{2\{X_2/X\}}]^{\mathfrak{R}}_{\mathfrak{D}})\,, \\ & \quad \mathtt{expand}([\alpha_3]) \bowtie \mathtt{expand}([\mu X_2.\,\gamma_2 \cup [\alpha_2]^{\mathfrak{R}_7}_{\mathfrak{D}_7}, \alpha_4])\,, \\ & \quad \mathtt{expand}([\alpha_4]) \bowtie \mathtt{expand}([\mu X_1.\,\gamma_1 \cup [\alpha_1]^{\mathfrak{R}_6}_{\mathfrak{D}_6}, \alpha_3])\,] \\ & \text{when } (type(\gamma_1) \cap type(\gamma_2)) \cap (\mathfrak{D}_6 \cup \mathfrak{D}_7) = \emptyset \\ & \text{and } type(\gamma_1)\backslash type(\gamma_2) \cap \mathfrak{R}_7 = \emptyset \text{ and } type(\gamma_2)\backslash type(\gamma_1) \cap \mathfrak{R}_6 = \emptyset \\ \bullet & [\,\mathtt{expand}([\mu X_1.\,\gamma_1 \cup [\alpha_1]^{\mathfrak{R}_6}_{\mathfrak{D}_6}, \alpha_3]) \bowtie \mathtt{expand}([\mu X_2.\,\gamma_2 \cup [\alpha_2]^{\mathfrak{R}_7}_{\mathfrak{D}_7}, \alpha_4])\,, \\ & \quad \mathtt{expand}([\alpha_3]) \bowtie \mathtt{expand}([\mu X_2.\,\gamma_2 \cup [\alpha_2]^{\mathfrak{R}_7}_{\mathfrak{D}_7}, \alpha_4])\,, \\ & \quad \mathtt{expand}([\alpha_4]) \bowtie \mathtt{expand}([\mu X_1.\,\gamma_1 \cup [\alpha_1]^{\mathfrak{R}_6}_{\mathfrak{D}_6}, \alpha_3])\,] \text{ otherwise} \end{cases}$$

where $\mathfrak{D} = \mathfrak{D}_6 \cup \mathfrak{D}_7$ and $\mathfrak{R} = \mathfrak{R}_6 \cup \mathfrak{R}_7$ 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.

Figure 12: Merging fixpoint operation nodes.

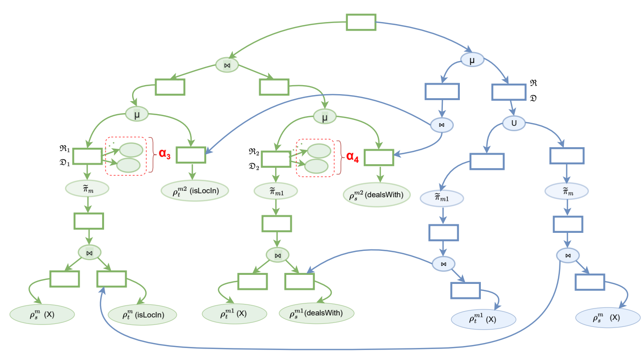

Figure 13: RLQDAG structure after merging recursions.

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.

Transformations of non-regular expressions. Notice that RLQDAG transformations support expressions with non-regular recursive patterns. For instance, A^nB^n can simply be written as the RLQDAG fixpoint operation node $\mu X.[A/B] \cup [A/X/B]$ where "/" is an abbreviation: W/Z stands for the RLQDAG subexpression that joins the target column of W with the source column of Z. As such,

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

$$\begin{cases} \bullet & [\,unpushed,\ pushed\,] \text{ when } \ a \notin \mathfrak{R} \text{ and } rep \text{ is false} \\ \bullet & [\,pushed\,] \text{ when } \ a \notin \mathfrak{R} \text{ and } rep \text{ is true} \\ \bullet & [\,unpushed\,] \text{ otherwise} \end{cases}$$

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

$$\begin{cases} \bullet & [\,unpushed_2,\ pushed,\ expand_{\alpha_2}\,] \text{ when } \ a \notin \mathfrak{R} \text{ and } rep \text{ is false} \\ \bullet & [\,pushed,\ expand_{\alpha_2}\,] \text{ when } \ a \notin \mathfrak{R} \text{ and } rep \text{ is true} \\ \bullet & [\,unpushed_2\,,\ expand_{\alpha_2}\,] \text{ otherwise} \end{cases}$$

where:

$$\begin{aligned} unpushed &= \widetilde{\pi}_a(\mathtt{expand}([\mu X.\,\gamma \cup [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}])) \\ pushed &= \mu X'.\,\mathtt{expand}([\widetilde{\pi}_a(\gamma)]) \cup [\mathtt{expand}(\alpha_{\{X/X'\}})]^{\mathfrak{R}}_{\mathfrak{D}} \\ unpushed_2 &= \widetilde{\pi}_a(\mathtt{expand}([\mu X.\,\gamma \cup [\alpha]^{\mathfrak{R}}_{\mathfrak{D}}, \alpha_2])) \\ expand_{\alpha_2} &= \mathtt{expand}([\widetilde{\pi}_a([\alpha_2])]) \end{aligned}$$

Figure 14: Pushing antiprojection in an equivalence node containing a fixpoint.

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

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

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

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

Figure 15: Pushing antijoin in an equivalence node containing a fixpoint.

transformations are equally applicable for regular and non-regular fixpoint operation nodes.

4.2 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' containing all the terms obtained by transformations. The `expand()` function is defined as follows:

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

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

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

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

$$
\mathrm{pfj}(\big[\sigma_f(\big[\gamma_1 \bowtie \gamma_2, \alpha\big])\big]) =
\begin{cases}
\bullet \quad \big[\, \mathtt{exp}(\big[\sigma_f(\gamma_1)\big]) \bowtie \mathtt{exp}(\gamma_2),\ \mathtt{exp}(\big[\sigma_f(\alpha)\big]) \,\big] \\
\quad\quad \textit{when filt(f)} \subseteq \textit{type}(\gamma_1) \wedge \textit{filt(f)} \nsubseteq \textit{type}(\gamma_2) \\
\bullet \quad \big[\, \mathtt{exp}(\gamma_1) \bowtie \mathtt{exp}(\big[\sigma_f(\gamma_2)\big]),\ \mathtt{exp}(\big[\sigma_f(\alpha)\big]) \,\big] \\
\quad\quad \textit{when filt(f)} \nsubseteq \textit{type}(\gamma_1) \wedge \textit{filt(f)} \subseteq \textit{type}(\gamma_2) \\
\bullet \quad \big[\, \mathtt{exp}(\big[\sigma_f(\gamma_1)\big]) \bowtie \mathtt{exp}(\big[\sigma_f(\gamma_2)\big]),\ \mathtt{exp}(\big[\sigma_f(\alpha)\big]) \,\big] \\
\quad\quad \textit{when filt(f)} \subseteq \textit{type}(\gamma_1) \wedge \textit{filt(f)} \subseteq \textit{type}(\gamma_2) \\
\bullet \quad \big[\, \sigma_f(\mathtt{exp}(\big[\varphi \bowtie \psi\big]),\ \mathtt{exp}(\big[\sigma_f(\alpha)\big])) \,\big]\ \textit{otherwise}
\end{cases}
$$

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

antiprojections in a join operation node, jassoc() for join associativity, dju() for distributivity of join over unions, etc.

$$\mathtt{allCodd}(\big[d\big]) \quad = \quad \mathrm{pfj}(\big[d\big]) \cup \mathrm{paj}(\big[d\big]) \cup \mathrm{jassoc}(\big[d\big]) \cup \ldots \cup \mathrm{dju}(\big[d\big])$$

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

4.3 Correctness and completeness

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

Proof. The proof is done by decomposition and induction. It is available in Appendix A of the extended version[3]. ◻

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 4, we consider $\big[\alpha\big]$ =* `unfold(expand`$_R$`(`$\big[\alpha'\big]$`))` *where α' is a consistent RLQDAG, and* `expand`$_R$`()` *is the* `expand()` *function in which rules in R are activated. The following properties hold:*

Property 1 (All pushable filters have been pushed).
$\forall\ \sigma_f(\gamma) \in \big[\alpha\big], \nexists d \in \gamma \mid d = \mu X.\ \big[\kappa\big] \cup \big[\alpha_2\big]^{\mathfrak{R}}_{\mathfrak{D}}$ *and* $\mathrm{filt(f)} \cap \mathfrak{D} = \emptyset$.

Property 2 (All pushable antiprojections pushed).
$\forall\ \widetilde{\pi}_a(\gamma) \in \big[\alpha\big], \nexists d \in \gamma \mid d = \mu X.\ \big[\kappa\big] \cup \big[\alpha_2\big]^{\mathfrak{R}}_{\mathfrak{D}}$ *and* $a \notin \mathfrak{R}$.

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

Property 4 (All mergeable fixpoints merged).
$\forall\ (\gamma_1 \bowtie \gamma_2) \in \big[\alpha\big]$, *if* $\mu X_1.\ \big[\kappa_1\big] \cup \big[\alpha_1\big]^{\mathfrak{R}_6}_{\mathfrak{D}_6} \in \gamma_1$ *and* $\mu X_2.\ \big[\kappa_2\big] \cup \big[\alpha_2\big]^{\mathfrak{R}_7}_{\mathfrak{D}_7} \in \gamma_2$ *and we have* $\gamma_1 \neq \gamma_2$ *and* $(\mathrm{type}(\gamma_1) \cap \mathrm{type}(\gamma_2)) \cap (\mathfrak{D}_6 \cup \mathfrak{D}_7) = \emptyset$ *and* $\mathrm{type}(\gamma_1) \backslash \mathrm{type}(\gamma_2) \cap \mathfrak{R}_7 = \emptyset$ *and* $\mathrm{type}(\gamma_2) \backslash \mathrm{type}(\gamma_1) \cap \mathfrak{R}_6 = \emptyset$ *then there exists* $d \in \big[\alpha\big]$ *such that* $d = \mu X.\ \big[\big[\kappa_1\big] \bowtie \big[\kappa_2\big]\big] \cup \big[\big[\alpha_1\big] \cup \big[\alpha_2\big]\big]^{\mathfrak{R}}_{\mathfrak{D}}$.

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

[3] https://doi.org/10.48550/arXiv.2312.02572

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 `expand`$_R$`()` leaves no room for such a missed opportunity, thus (iv) leading to a contradiction. ◻

4.4 Implementation

We implemented the RLQDAG in Scala. It takes as input a recursive query, generates its plan space, selects a plan with best estimated cost, and then send it to PostgreSQL for evaluation.

The system implementation is organized in two modules: the `enumerator` that computes the RLQDAG expansion, and the `cost estimator` in charge of (i) annotating equivalence nodes with estimated cardinalities (using base relations data statistics), and (ii) operation nodes with estimated costs (as described in [36, 37]).

Our experiments focus on the evaluation of the efficiency of plan enumeration and the quality of the generated plan spaces. Therefore, we first expand, and then estimate costs.

Plan exploration is achieved by an implementation of the `expand()` function presented in Section 4.2. This implementation relies on a top-down traversal of the RLQDAG structure using two nested loops and recursive calls to `expand()`. The purpose of the loops is to efficiently traverse the RLQDAG structure in order to trigger a rule whenever it is applicable. In order to test rule applicability, we use Scala's pattern matching on the structure exposed by the nested loops. There is thus a direct correspondence between the implementation and the formal presentation of the rules that use syntactic case by case decompositions. In the implementation, the application of rules follows an order determined by the depth-first exploration of the RLQDAG's structure and the implicit order of operation nodes within equivalent nodes (ordered sets in Scala). In-memory unicity of operation and equivalence nodes is ensured. At each step of the expansion, equivalence nodes are unified as early as possible, so that duplicates are never created. Without timeout, since RLQDAG terms are finite and all recursive calls are done on strictly smaller subterms, the expansion always terminates. Furthermore, all considered rules are deterministic and all applicable rules are always applied, in all possible sequences of application. Therefore, for a given RLQDAG, the result of the expansion is always the same: the expansion is deterministic. In general, filtering expressions (predicates) can be arbitrarily complex. Filter pushdown thus requires a cost estimation in order to decide which is the best plan between the initial and the rewritten term. This is why, by default, rules pf() and pp() always preserve the initial term in the expansion (i.e. by default, *rep* = false).

When filtering expressions are very simple (e.g. they can be evaluated in constant time), it makes sense to consider only terms in which filtering expressions are pushed. This simple heuristic (enabled by setting *rep* = true), used in experiments, allows to discard suboptimal plans during plan space exploration.

5 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 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.

5.1 Experimental setup

Datasets. We consider various unmodified third-party datasets, graphs and trees, real and generated, as described in Table 1.

Table 1: Datasets (available from [56]).

Dataset	#nodes & #edges	Type	Nature
Yago [64]	42M & 62M	Knowledge graph	Real
Bahamas Leaks [8]	202K & 249K	Property graph	Real
Airbnb [6, 50]	24K & 14K	Property graph	Real
LDBC [11]	908K & 1.9M	Property graph	Synthetic
Wikitree [19]	1.3M & 9.1M	Tree	Real
Academic tree [38]	765K & 1.5M	Tree	Real

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[4], 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 [50]. 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 and Academic Tree datasets. Queries and datasets used in experiments are available at [56].

Hardware and environment. All experiments are conducted on a machine with an Intel Xeon Silver 4114 2.20GHz CPU, 192GB of RAM and 4TB 7200rpm SATA disks. The machine runs Ubuntu 20.4, Java 8 and the Scala 2.11 compiler with default parameters.

5.2 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 2, 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 the RLQDAG implementation with the performance of MuEnum. Figures 17, 18, 19, and Fig. 20 respectively show the results obtained for the queries over each dataset.

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

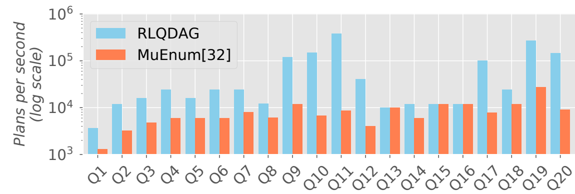

Figure 17: Yago queries.

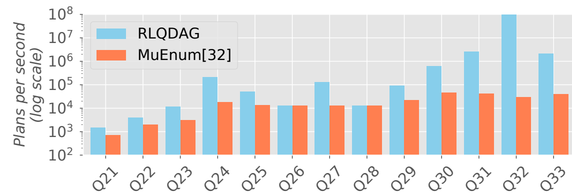

Figure 18: Bahamas Leaks queries.

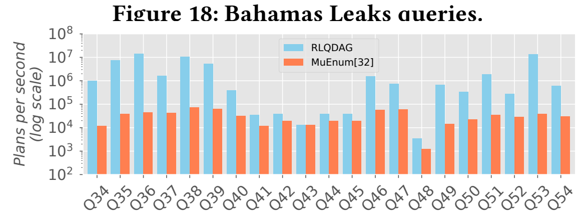

Figure 19: Airbnb and LDBC queries.

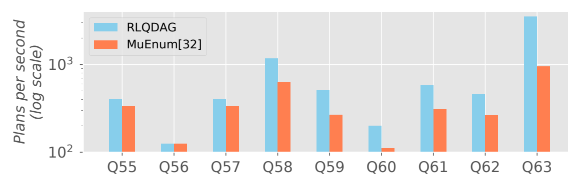

Figure 20: Non-regular queries.

enumerates plans much faster (up to two orders of magnitude) when compared to MuEnum[5].

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 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

[4] We consider 7 queries (Q1-Q7) taken from [3], 2 queries (Q8-Q9) taken from [65], (Q10-Q11) taken from [28] and (Q12-Q20) come from [32].

[5] 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.

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).

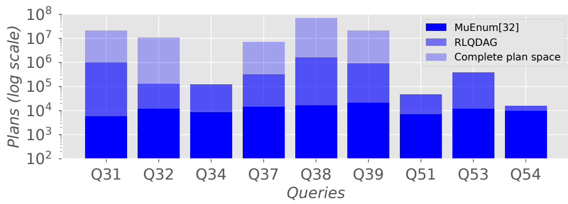

Figure 21: Plan spaces explored in 500ms.

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).

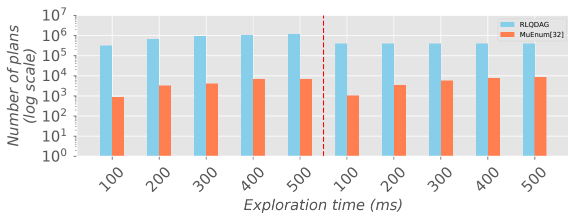

Figure 22: Plans explored per time budgets for Q31 and Q53.

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.

Scalability with increasing query complexity. We now assess to which extent the approach scales with query complexity. For that purpose, we consider a notion of recursive query complexity as the number of joined transitive closure relations. This slightly extends the usual notion of query complexity (traditionally measured as the number of joined relations) found in the literature to also encompass recursion. This accounts for the fact that both joins and recursions are complex operations, and also for the fact that they both significantly contribute to plan space size increase due to combinatorial explosions of equivalent plans (join ordering combined with rearranged recursions produce many more equivalent plans). Specifically, we consider a variable-length query which is a concatenation of transitive closures of relations: $Q_{r_i} = a_1^+/a_2^+/.../a_i^+$ for a given length i. An increment in the query length increases both the number of recursions and joins, thus the query complexity, and therefore the size of the complete plan space. Specifically, Q_{r_i} contains i recursions and $2i - 1$ joins ($i - 1$ joins at top level plus one join within each recursive part).

Figure 23 shows a comparative analysis of plan enumeration speed of RLQDAG and MuEnum when the query complexity increases. It shows the number of plans produced per millisecond (on the y axis, in log scale) for a given query length (on the x axis). Curves represent plan exploration speed. We observe that RLQDAG's plan enumeration is always faster. Even more importantly, for the RLQDAG we observe an acceleration in enumeration, i.e. an increase of enumeration speed with respect to query complexity. The more complex is the query, the bigger are the complete plan space sizes, and the more RLQDAG becomes effective. This is because RLQDAG transforms sets of terms of increasing sizes, which explains the progressive acceleration. MuEnum explores the plan space term by term and hits a maximal exploration speed. In comparison, RLQDAG's sharing and set-based exploration are clearly more effective. For example, for Q_{r_8} that contains 8 recursions and 7 top-level joins: MuEnum explored 242K plans with a timeout of 10s, whereas RLQDAG explored 45M plans (2 orders of magnitude more).

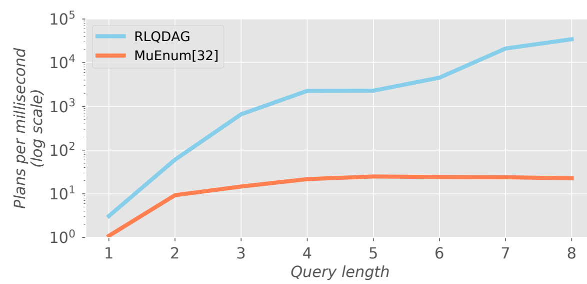

Figure 23: Enumeration with increasing query complexity.

We now assess how relevant are faster explorations of larger plan spaces in terms of query evaluation performance.

5.3 Relevance of large plan space exploration

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 cost estimation [37], thus making relevant a head to head comparison. We measure and compare the times spent by PostgreSQL for evaluating the plans.

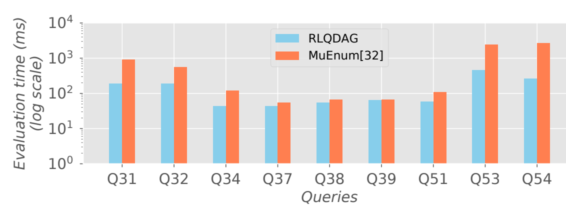

Figure 24: Evaluating best estimated plans of spaces of Fig 21.

For example, Fig. 24 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.

Query running time with increasing exploration time budget. For a given query, we now inspect the impact on performance of increasing time budgets for plan space exploration. For that purpose, we trigger the plan space explorations with RLQDAG and `MuEnum` for different time budgets ranging from 100ms to 500ms. We then measure the time spent in evaluating the best estimated plans taken from the corresponding plan spaces. We compare their respective performances. Fig. 25 shows the results obtained for queries Q31 and Q53. The y axis shows the time spent (in log scale) in evaluating the best estimated plan obtained within the time budget shown on the x axis. The sizes of corresponding plan spaces are given in Fig. 22.

For Q31, we observe that for both systems, the greater the exploration budget, the more efficient is the evaluation. We also observe that RLQDAG makes it possible to obtain more efficient plans much sooner (i.e. already with much smaller exploration time budgets). For Q53, RLQDAG is even more decisive since it explores the complete plan space in 100ms. The best possible estimated plan is thus already obtained, not needing any additional exploration budget. In comparison, increasing time budgets for `MuEnum` allows it to explore only small fractions of the whole plan space (as shown in Fig. 21), failing to obtain a plan with better performance.

Overall, best estimated terms selected from larger plan spaces are more efficient. For any given exploration time budget, RLQDAG always produces more efficient terms than `MuEnum`. This shows that in practice, larger plan spaces are very prone to contain more efficient recursive plans. This confirms the importance of efficiently exploring large recursive plan spaces.

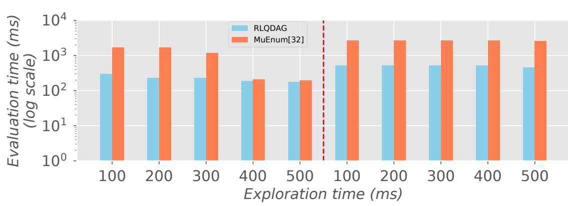

Figure 25: Evaluating best estimated plans of spaces of Fig 22.

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 [58], Neo4j [62] version 4.4.11, Blazegraph [55] version 2.1.6), and 3 plain-vanilla RDBMS capable of evaluating recursive SQL queries (PostgreSQL [52] version 15.1, MySQL [1] version 8.1.0, SQLite [30] version 3.36.0), and one state-of-the-art Datalog engine (Soufflé [33] version 2.4.1). 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. 26 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. 26 for the given query.

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

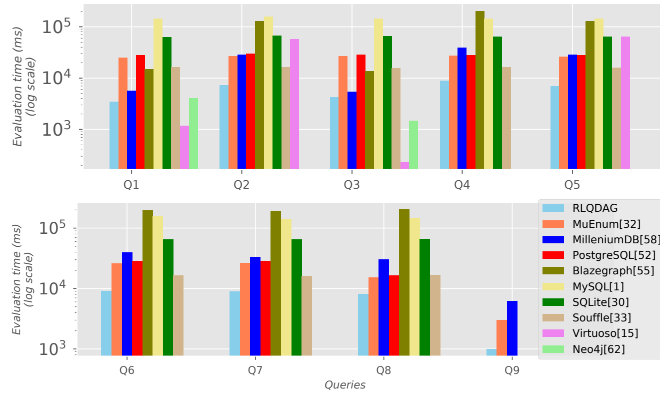

Figure 26: Comparative evaluation of third-party queries.

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. Likewise, Datalog engines like Soufflé are unable to generate many of the evaluation plans generated by the RLQDAG. For example they cannot merge recursions (as done in Section 4.1).

The plan spaces theoretically producible by RLQDAG and `MuEnum` are the same. However, `MuEnum` explores them term by term whereas RLQDAG explores them in a grouped manner which is much faster. This is because the compact representation of RLQDAG allows to transform sets of terms at once. As a consequence, RLQDAG makes it possible to explore in practice much larger portions of the theoretical plan space for the same time budget. Since transformations presented in Section 4.1 are fully compositional, plans in which recursions are merged open further optimization opportunities. They unlock the exploration of many more plans in which for example joins and filters are pushed within merged recursions, etc. Such plans often happen to be much more efficient in practice.

6 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. The proposed formalisation of the RLQDAG in terms of syntax and semantics provided a convenient – if not instrumental – means to develop transformations. It helps in defining expansions, and for detecting and fixing intricate transformational issues. We believe that this formalization can also serve in further extensions (such as groupBy and aggregations in the presence of recursion), thus contributing to the extensibility of the top-down transformational approach. 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] 2023. MySQL. https://www.mysql.com/.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases: The Logical Level* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., USA.

[3] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Shadi Ghajar-Khosravi, and Mark H Chignell. 2017. 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.2017.47*. 470–473.

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

[5] Alfred V. Aho and Jeffrey D. Ullman. 1979. Universality of Data Retrieval Languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas) *(POPL '79)*. ACM, New York, NY, USA, 110–119. https://doi.org/10.1145/567752.567763

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

[7] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford,UK, March 16-19, 2010. Revised Selected Papers (Lecture Notes in Computer Science)*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers (Eds.), Vol. 6702. Springer Berlin Heidelberg, Berlin, Heidelberg, 262–281. https://doi.org/10.1007/978-3-642-24206-9_16

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

[9] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. 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* (Cambridge, Massachusetts, USA) *(PODS '86)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/6012.15399

[10] Francois Bancilhon and Raghu Ramakrishnan. 1986. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) *(SIGMOD '86)*. Association for Computing Machinery, New York, NY, USA, 16–52. https://doi.org/10.1145/16894.16859

[11] Peter Boncz. 2013. LDBC: Benchmarks for Graph and RDF Data Management. In *Proceedings of the 17th International Database Engineering ; Applications Symposium* (Barcelona, Spain) *(IDEAS '13)*. Association for Computing Machinery, New York, NY, USA, 1–2. https://doi.org/10.1145/2513591.2527070

[12] Surajit Chaudhuri. 1998. An Overview of Query Optimization in Relational Systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, Washington, USA) *(PODS '98)*. Association for Computing Machinery, New York, NY, USA, 34–43. https://doi.org/10.1145/275487.275492

[13] Surajit Chaudhuri and Kyuseok Shim. 1994. Including group-by in query optimization. In *VLDB*, Vol. 94. 12–15.

[14] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (June 1970), 377–387. https://doi.org/10.1145/362384.362685

[15] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Eng. Bull.* 35 (2012), 3–8.

[16] Zhiwei Fan, Jianqiao Zhu, Zuyu Zhang, Aws Albarghouthi, Paraschos Koutris, and Jignesh M. Patel. 2019. Scaling-Up In-Memory Datalog Processing: Observations and Techniques. *Proc. VLDB Endow.* 12, 6 (2019), 695–708. https://doi.org/10.14778/3311880.3311886

[17] Pit Fender and Guido Moerkotte. 2013. Counter Strike: Generic Top-down Join Enumeration for Hypergraphs. *Proc. VLDB Endow.* 6, 14 (sep 2013), 1822–1833. https://doi.org/10.14778/2556549.2556565

[18] Pit Fender and Guido Moerkotte. 2013. Top down plan generation: From theory to practice. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 1105–1116. https://doi.org/10.1109/ICDE.2013.6544901

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

[20] Matthew Francis-Landau, Tim Vieira, and Jason Eisner. 2020. Evaluation of Logic Programs with Built-Ins and Aggregation: A Calculus for Bag Relations. *CoRR* abs/2010.10503 (2020), 1–27. arXiv:2010.10503 https://arxiv.org/abs/2010.10503

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

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

[23] Georges Gardarin. 1987. Magic Functions: A Technique to Optimize Extended Datalog Recursive Programs. 21–30.

[24] Roy Goldman and Jennifer Widom. 1997. *Dataguides: Enabling query formulation and optimization in semistructured databases*. Technical Report. Stanford.

[25] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf

[26] Goetz Graefe and David J. DeWitt. 1987. The EXODUS Optimizer Generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '87)*. ACM, New York, NY, USA, 160–172. https://doi.org/10.1145/38713.38734

[27] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *Proceedings of the Ninth International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 209–218. http://dl.acm.org/citation.cfm?id=645478.757691

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

[29] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. 1989. Extensible Query Processing in Starburst. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 377–388. https://doi.org/10.1145/67544.66962

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

[31] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1213–1216. https://doi.org/10.1145/1989323.1989456

[32] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. 2020. On the Optimization of Recursive Relational Queries: Application to Graph Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 681–697. https://doi.org/10.1145/3318464.3380567

[33] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science)*, Swarat Chaudhuri and Azadeh Farzan (Eds.), Vol. 9780. Springer, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

[34] Michael Kifer and Eliezer L. Lozinskii. 1990. On Compile-Time Query Optimization in Deductive Databases by Means of Static Filtering. *ACM Trans. Database Syst.* 15, 3 (sep 1990), 385–426. https://doi.org/10.1145/88636.87121

[35] André Koschmieder and Ulf Leser. 2012. Regular path queries on large graphs. In *International Conference on Scientific and Statistical Database Management*. Springer, 177–194.

[36] Muideen Lawal. 2021. *On Cost Estimation for the Recursive Relational Algebra*. Theses. Université Grenoble Alpes [2020-....]. https://theses.hal.science/tel-03322720

[37] Muideen Lawal, Pierre Genevès, and Nabil Layaïda. 2020. A Cost Estimation Technique for Recursive Relational Algebra. *Proceedings of the 29th ACM International Conference on Information & Knowledge Management* (2020), 3297–3300.

[38] Jean F Liénard, Titipat Achakulvisut, Daniel E Acuna, and Stephen V David. 2018. Intellectual synthesis in mentorship determines success in academic careers. *Nature communications* 9, 1 (2018), 1–13.

[39] Guy M. Lohman. 1988. Grammar-like Functional Rules for Representing Query Optimization Alternatives. *SIGMOD Rec.* 17, 3 (June 1988), 18–27. https://doi.org/10.1145/971701.50204

[40] Guy M. Lohman, C. Mohan, Laura M. Haas, Dean Daniels, Bruce G. Lindsay, Patricia G. Selinger, and Paul F. Wilms. 1985. *Query Processing in R**. Springer Berlin Heidelberg, Berlin, Heidelberg, 31–47. https://doi.org/10.1007/978-3-642-82375-6_2

[41] Guido Moerkotte, Pit Fender, and Marius Eich. 2013. On the correct and complete enumeration of the core search space. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 493–504.

[42] J. F. Naughton, R. Ramakrishnan, Y. Sagiv, and J. D. Ullman. 1989. Efficient Evaluation of Right-, Left-, and Multi-Linear Rules. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 235–242. https://doi.org/10.1145/67544.66948

[43] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. 2000. Efficient and Extensible Algorithms for Multi Query Optimization. *SIGMOD Rec.* 29, 2 (May 2000), 249–260. https://doi.org/10.1145/335191.335419

[44] Domenico Saccà and Carlo Zaniolo. 1985. 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* (Cambridge, Massachusetts, USA) *(PODS '86)*. Association for Computing Machinery, New York, NY, USA, 16–23. https://doi.org/10.1145/6012.6013

[45] Maximilian Schleich, Amir Shaikhha, and Dan Suciu. 2023. Optimizing Tensor Programs on Flexible Storage. *Proc. ACM Manag. Data* 1, 1 (2023), 37:1–37:27. https://doi.org/10.1145/3588717

[46] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. ACM, New York, NY, USA, 23–34.

https://doi.org/10.1145/582095.582099

[47] Jiwon Seo, Stephen Guo, and Monica S. Lam. 2015. SociaLite: An Efficient Graph Query Language Based on Datalog. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1824–1837. https://doi.org/10.1109/TKDE.2015.2405562

[48] Anil Shanbhag and S. Sudarshan. 2014. Optimizing Join Enumeration in Transformation-Based Query Optimizers. *Proc. VLDB Endow.* 7, 12 (aug 2014), 1243–1254. https://doi.org/10.14778/2732977.2732997

[49] Leonard Shapiro, David Maier, Paul Benninghoff, Keith Billings, Yubo Fan, Kavita Hatwal, Quan Wang, Yu Zhang, H.-M Wu, and Bennet Vance. 2001. Exploiting Upper and Lower Bounds in Top-Down Query Optimization. 20 – 33. https://doi.org/10.1109/IDEAS.2001.938068

[50] Chandan Sharma, Roopak Sinha, and Kenneth Johnson. 2021. Practical and comprehensive formalisms for modelling contemporary graph query languages. *Information Systems* 102 (2021), 101816. https://doi.org/10.1016/j.is.2021.101816

[51] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1135–1149. https://doi.org/10.1145/2882903.2915229

[52] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of POSTGRES *(SIGMOD '86)*. Association for Computing Machinery, New York, NY, USA, 340–355.

[53] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2011. Equality Saturation: A New Approach to Optimization. *Log. Methods Comput. Sci.* 7, 1 (2011), 1–37. https://doi.org/10.2168/LMCS-7(1:10)2011

[54] K. Tuncay Tekle and Yanhong A. Liu. 2011. More Efficient Datalog Queries: Subsumptive Tabling Beats Magic Sets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (Athens, Greece) *(SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 661–672. https://doi.org/10.1145/1989323.1989393

[55] Bryan Thompson, Mike Personick, and Martyn Cutcher. 2016. The Bigdata® RDF graph database. In *Linked Data Management*. Chapman and Hall/CRC, 221–266.

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

[57] Victor Vianu. 2021. Datalog Unchained. In *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Virtual Event, China, June 20-25, 2021*, Leonid Libkin, Reinhard Pichler, and Paolo Guagliardo (Eds.). ACM, 57–69. https://doi.org/10.1145/3452021.3458815

[58] Domagoj Vrgoč, Carlos Rojas, Renzo Angles, Marcelo Arenas, Diego Arroyuelo, Carlos Buil-Aranda, Aidan Hogan, Gonzalo Navarro, Cristian Riveros, and Juan Romero. 2022. MillenniumDB: An Open-Source Graph Database System. *Data Intelligence* (2022), 1–39.

[59] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. 2015. Asynchronous and Fault-Tolerant Recursive Datalog Evaluation in Shared-Nothing Engines. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1542–1553. https://doi.org/10.14778/2824032.2824052

[60] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 11 (2020), 1919–1932. http://www.vldb.org/pvldb/vol13/p1919-wang.pdf

[61] Yisu Remy Wang, Mahmoud Abo Khamis, Hung Q Ngo, Reinhard Pichler, and Dan Suciu. 2022. Optimizing Recursive Queries with Program Synthesis. *arXiv preprint arXiv:2202.10390* (2022), 1–23.

[62] Jim Webber. 2012. A Programmatic Introduction to Neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, Arizona, USA) *(SPLASH '12)*. Association for Computing Machinery, New York, NY, USA, 217–218. https://doi.org/10.1145/2384716.2384777

[63] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434304

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

[65] Nikolay Yakovets, Parke Godfrey, and Jarek Gryz. 2015. WAVEGUIDE: Evaluating SPARQL Property Path Queries.. In *EDBT*, Vol. 2015. 525–528.

[66] Harald Zauner, Benedikt Linse, Tim Furche, and François Bry. 2010. A RPL through RDF: Expressive navigation in RDF graphs. In *International Conference on Web Reasoning and Rule Systems*. Springer, 251–257.