THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Sur l'optimisation de l'énumération de plans récursifs avec une application aux requêtes de graphes de propriétés

On the Optimization of Recursive Plan Enumeration with an Application to Property Graph Queries

Présentée par :

AMELA FEJZA

Direction de thèse :

Pierre GENEVES   Directeur de thèse

Rapporteurs :

Mohand-Saïd HACID
PROFESSEUR, Université Claude Bernard Lyon 1
Ladjel BELLATRECHE
PROFESSEUR, École Nationale Supérieure de Mécanique et dAérotechnique

Thèse soutenue publiquement le   ???   devant le jury composé de :

Pierre GENEVES	Directeur de thèse
DIRECTEUR DE RECHERCHE, CNRS	
Mohand-Saïd HACID	Rapporteur
PROFESSEUR, Université Claude Bernard Lyon 1	
Ladjel BELLATRECHE	Rapporteur
PROFESSEUR, École Nationale Supérieure de Mécanique et dAérotechnique	
Sophie DUPUY-CHESSA	Examinatrice
PROFESSEUR, Université Grenoble Alpes	
Stefania Gabriela DUMBRAVA	Examinatrice
MAITRE DE CONFERENCE, École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise & Institut Polytechnique de Paris	

Abstract

Graph data structures containing massive interrelated data are omnipresent nowadays. Recursive queries constitute a powerful means to extract valuable information from graphs. For example, by enabling navigation along edge sequences of arbitrary length, regular path queries make it possible to analyze relations between distant graph entities. However, recursion often makes query answering very expensive, sometimes even hardly feasible in practice. Methods for recursive query optimization are crucial. The difficulty of the query optimization problem in the presence of recursion is notoriously known.

This thesis describes the theoretical and practical foundations of the *recursive logical query directed acyclic graph* (RLQDAG) for efficiently enumerating recursive query plans in transformation-based query optimizers; and an application for extracting information from property graphs. The RLQDAG extends earlier techniques developed for recursion-free queries and recent developments in recursive relational algebra, for the purpose of optimizing the plan enumeration phase in transformation-based query optimizers. This phase is crucial as it may produce plans which are drastically more efficient, with a direct impact on the feasibility and efficiency of recursive query answering in practice.

This dissertation starts with a thorough litterature review on query languages for graphs, methods for query optimization, and their limits in the presence of recursion. Then, in a contribution part, the RLQDAG is introduced by formalizing and extending important concepts such as subterm-sharing in order to capture and enable grouped transformations of recursive query plans. A subsequent Chapter studies the application of the RLQDAG for recursive query answering with property graphs. A prototype implementation of the RLQDAG is shown to provide significant performance gains compared to the state-of-the-art.

Résumé

Les graphes contenant des masses d'information interreliées sont des structures de données omniprésentes de nos jours. Les requêtes récursives constituent un moyen puissant pour extraire des informations potentiellement précieuses de ces graphes. Par exemple, en permettant la navigation le long de séquences d'arêtes de longueur arbitraire, les requêtes de chemins réguliers permettent d'analyser les relations entre des entités distantes du graphe. Cependant, la récursivité rend souvent la réponse aux requêtes très coûteuse, parfois même difficilement réalisable en pratique. Les méthodes d'optimisation des requêtes récursives sont cruciales. La difficulté du problème de l'optimisation des requêtes en présence de récursion est notoirement connue.

Cette thèse décrit les fondements théoriques et pratiques du *graphe acyclique dirigé logique récursif de requêtes* (RLQDAG) pour énumérer efficacement les plans de requêtes récursifs dans les optimiseurs de requêtes basés sur le principe de transformation ; ainsi qu'une application pour extraire de l'information de graphes de propriétés. Le RLQDAG étend les techniques précédemment développées pour les requêtes non récursives ainsi que les derniers développements en algèbre relationnelle récursive, dans le but d'optimiser la phase d'énumération de plans dans les optimiseurs de requêtes. Cette phase est cruciale car elle peut produire des plans d'évaluation qui sont drastiquement plus efficaces, avec un impact direct sur la faisabilité et l'efficacité de l'évaluation de requêtes récursives en pratique.

Cette thèse commence par une revue approfondie de la littérature sur les langages de requêtes pour les graphes, les méthodes d'optimisation de requêtes et leurs limites en présence de récursion. Ensuite, dans une partie de contribution, le RLQDAG est introduit en formalisant et en étendant des concepts importants tels que le partage de sous-termes afin de capturer et de permettre des transformations groupées de sous-plans de requêtes récursives. Un chapitre suivant étudie l'application du RLQDAG pour l'évaluation de requêtes récursives avec des graphes de propriétés. Un prototype d'implémentation du RLQDAG permet d'obtenir des gains de performance significatifs par rapport à l'état de l'art.

Remerciements

Je tiens à remercier mon encadrant de thèse Pierre Genevès pour l'encadrement et la disponibilité tout au long de cette thèse. Je lui suis profondément reconnaissante pour m'avoir poussé au mieux de mes capacités, pour avoir toujours cru en moi, et pour m'avoir montré que la recherche peut aussi être cool. Je tiens également à remercier Nabil Layaïda pour son encouragement et son aide pendant cette thèse.

Je tiens ensuite à remercier tous les membres du jury: Mohand-Saïd Hacid et Ladjel Bellatreche, pour avoir accepté de rapporter mon manuscrit de thèse; Sophie Dupuy-Chessa et Stefania Dumbrava pour avoir accepté d'examiner mon manuscrit de thèse. Je suis honorée de votre participation.

Un grand merci à ceux avec qui j'ai partagé cette aventure de thèse: Sarah, pour sa positivité et l'intelligence; Muideen, pour sa modestie; Luisa, la grande sportive de notre équipe. Je voudrais aussi remercier Ugo, mon collègue de bureau pour avoir toujours été gentil, même quand le bureau est trop chaud pour une personne normale. Un grand merci à Laurent, pour toutes les conversations sur le foot; Nils et tous les stagiaires de l'équipe Tyrex de l'année dernière. Merci a tous les membres de l'équipe Tyrex pour l'ambiance et l'atmosphère de travail très agréable.

Je voudrais remercier Orestia, mon amie depuis collège que j'ai si rarement rencontrée ces dernières années, mais qui est toujours proche de mon cœur; Lina, pour l'amitié sans faille et pour être un exemple à suivre pour moi; Doni et Anto, mes compagnons de voyage qui étaient toujours intéressés par mon doctorat; Xheni, qui est devenu une amie très proche ces dernières années en France; tous mes amis albanais en France et tous mes amis et collègues que j'ai rencontrés pendant mes études de master et mon doctorat ici en France.

Finalement, je voudrais remercier mes parents et mes deux sœurs, Sarah et Aymes pour leur amour inconditionnel et leur soutien pas seulement tout au long de cette thèse, mais tout au long de ma vie. Je suis vraiment reconnaissante à mes parents de rendre possible pour moi de faire toujours les bons choix dans la vie. Je remercie tout particulièrement mes sœurs pour s'être occupées de moi ces deux dernières années, je sais que ce n'est pas facile de vivre avec une personne qui fait un doctorat.

Introduction

Recursive queries enable powerful information extraction, especially from linked data structures such as trees and graphs.

Graphs data structures containing huge amounts of information are ubiquitous nowadays, from social networks to bioinformatics, e-commerce, research on explainable artificial intelligence, etc. Graph datasets found in practice often contain valuable information to be extracted from interrelated data. Such information might be found by following not only direct links but indirect ones as well. i.e. by exploring sequences of links. This is where recursion comes in handy. Using recursive queries such as regular path queries, one can navigate in depth in the graph to extract valuable information between distant nodes. However, powerful recursive queries often come with a price: they are most often difficult to evaluate and are very prone to lead to inefficient information extraction. Depending on graph instances, the evaluation of recursive queries can be very costly or even infeasible in practice.

Most data management systems are designed to support primarily non-recursive queries, as also noticed in [Wang et al., 2022]. Many query optimizers are based on the transformation-based Volcano framework [Graefe and McKenna, 1993] which was designed specifically for optimizing non-recursive query plans. 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 [Abiteboul et al., 1995, Bancilhon and Ramakrishnan, 1986] resulted in powerful recursive relational algebras [Aho and Ullman, 1979, Agrawal, 1988, Jachiet et al., 2020], capable of capturing queries with transitive closures [Agrawal, 1988] and even more general forms of recursion [Aho and Ullman, 1979, Jachiet et al., 2020]. This line of works recently led to μ-RA [Jachiet et al., 2020] which provides a rich set of rewrite rules for recursive terms enabling efficient recursive evaluation plans not available with earlier approaches.

The enumeration phase is crucial as it may produce terms which are drastically more efficient. It has been heavily researched for recursion-free queries. It is common practice to allocate a time budget for this enumeration phase, as it is notoriously known that exhaustive plan space explorations may not be feasible in

practice for certain queries. The faster we generate the space of equivalent plans, the more likely we will be able to find terms with more efficient evaluation.

With recursion, plan spaces are often significantly larger than in the non-recursive setting, due to new interplays between recursive and non-recursive operators. The efficiency of recursive plan enumeration becomes critical. The speed of plan enumeration directly determines whether query evaluation plans enabled by, e.g. μ-RA [Jachiet et al., 2020], are within range or theoretically reachable, but still out of reach for a practical query optimizer. This motivates the search for efficient methods for enumerating recursive plans.

Thesis Contributions. The main contribution of this thesis is the recursive logical query dag (RLQDAG), which extends Volcano's LQDAG [Graefe and McKenna, 1993] and the μ-RA framework [Jachiet et al., 2020] for the purpose of efficiently enumerating recursive query plans. Contributions include the first extension of the LQDAG with the support of recursive terms; a formalization of important RLQDAG concepts in terms of formal syntax and semantics, with a particular focus on the sharing of common subterms in the presence of recursion; RLQDAG transformations that generalize rewrite rules for individual recursive terms of [Jachiet et al., 2020] for enabling efficient and grouped transformations of sets of recursive terms. The RLQDAG relies on a concept of annotated equivalence nodes with incremental updates, used for guiding transformations of recursive subterms. Contributions also include a complete prototype implementation of the proposed approach; the syntax of a high-level query language fragment (UCRPQ$_{PG}$) specifically designed for property graphs; its automated translation for generating an RLQDAG from a given query; and a complete experimental assessment of the whole approach using third-party regular path queries on synthetic and real datasets.

Thesis Outline. This dissertation is structured in two main parts: state-of-the-art and contribution.

The first part focuses on the state-of-the-art about graph data models and query optimizers. First, we present the most important graph data models and the most popular and richest query languages. Then, we investigate the different approaches used in query optimizers and review the attempts of extending them with recursion.

The second part focuses on the contribution. The RLQDAG is presented as an extension of the LQDAG widely used in query optimizers, with the support of recursion, and appropriate techniques to efficiently transform recursive terms. We also propose an extended query language fragment (a variant of UCRPQs) for property graphs that allows to support more expressive queries on these graphs. We define a syntax and show how this fragment can be translated into the RLQDAG. We report on practical experiments with a prototype implementation of the RLQDAG in a query optimizer. Finally we conclude and comment on several perspectives for further research opened by these works.

Contents

Part I

State of the art

Chapter 1

Graph data models and query languages

1.1 Graph data models

Nowadays, two main graph data models can be found in practice: the Resource Description Framework (RDF) and the property graph model (PG). There exist several query languages to manipulate the data formulated in each data model. In this section we first review the main differences between the two popular graph data models and then we discuss the corresponding query languages.

1.1.1 Knowledge graphs

Resource Description Framework (RDF) [Cyganiak et al., 2014] is a standard defined by the World Wide Web Consortium (W3C) that represents interconnected data. The RDF data model allows one to describe a network of entities and the relationships between them by expressing statements about resources. The most basic statement is expressed as a RDF triple: *(subject, predicate, object)*. Such a triple expresses the fact that the two resources *subject* and *object* are connected by the relation *predicate*. This relation is represented as a labelled directed edge, as illustrated in Figure 1.1. RDF defines the concept of *knowledge graph* as a set of such triples.
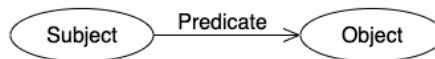
Figure 1.1: RDF example

The subject designates a resource and can be expressed as an IRI (Internationalized Resource Identifier) or as a blank node. An IRI is a string of characters that identifies a ressource. In addition to URI (Uniform Resource Identifier) scheme which allowed just ASCII characters, IRI scheme extends to any unicode character. An IRI has the form scheme:path and an example can be the following: "https://www.univ-grenoble-alpes.fr". It looks like an url, but it does not

necessarily need to be an internet accessible address. In RDF technology, data is accessed by using ontologies, which can be described as sets of rules encoded in RDF that model the relationships between objects. To author ontologies there exists the family of knowledge representation languages known as Web Ontology Language (OWL).

More formally, a knowledge graph can be defined as a pair (V, E), where V is a finite set of vertices and E is a finite set of labelled edges. Each edge $e \in E$ is labelled with a label taken from a finite set *Labelled* of labels. In other terms, $E \subset V \times Labelled \times V$.

An example of a knowledge graph is shown in Figure 1.2:
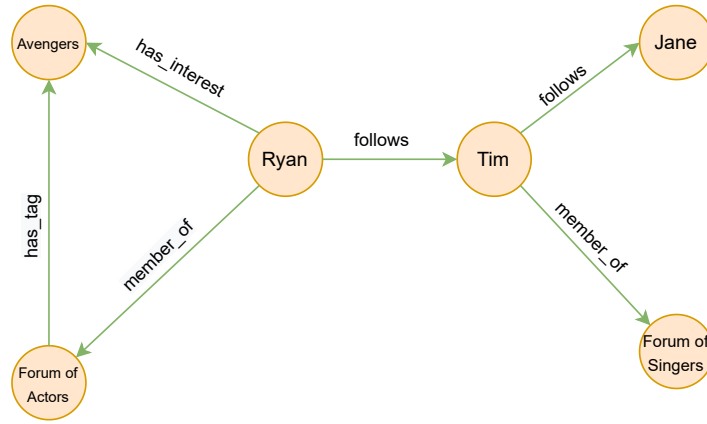
Figure 1.2: Knowledge graph example

A representation in a table for the graph example is illustrated in Figure 1.3.

src	label	trg
Jean	livesIn	Paris
France	isLocatedIn	Europe
Jim	worksAt	Inria
Jim	knows	Jean
...	...	...

Figure 1.3: A possible knowledge graph representation using tables.

An well-known knwoledge graph in practice that we will use later for experiments is Yago [YAGO, 2019], which includes general knowledge about people, movies, cities, countries, and organizations.

1.1.2   Property graphs

Property graphs constitute another type of graph data model where the relationships are also labelled, but can carry more information than in knowledge graphs. In property graphs both nodes and edges are labelled and each of them can be annotated with a list of key-value pairs. More formally, a property graph is represented as follows $(V, E, \beta, \eta, \gamma)$:

- V is a finite set of vertices.

- E is a finite set of edges.

- $\beta : E \to (V \times V)$ is a function that assigns to each edge the pair of vertices that constitute the source and target vertices.

- $\eta : V \cup E \to Labelled$ is a function that assigns to each vertex and edge an element in the finite set *Labelled*.

- $\gamma : (V \cup E) \times \mathcal{K} \to \mathcal{L}$ is a partial function that assigns a value to a given key of a vertex or an edge, where $\mathcal{K}$ is a set of property keys and $\mathcal{L}$ is a set of values.

An example of a property graph is illustrated in Figure 1.4. Each node and edge contains specific properties.
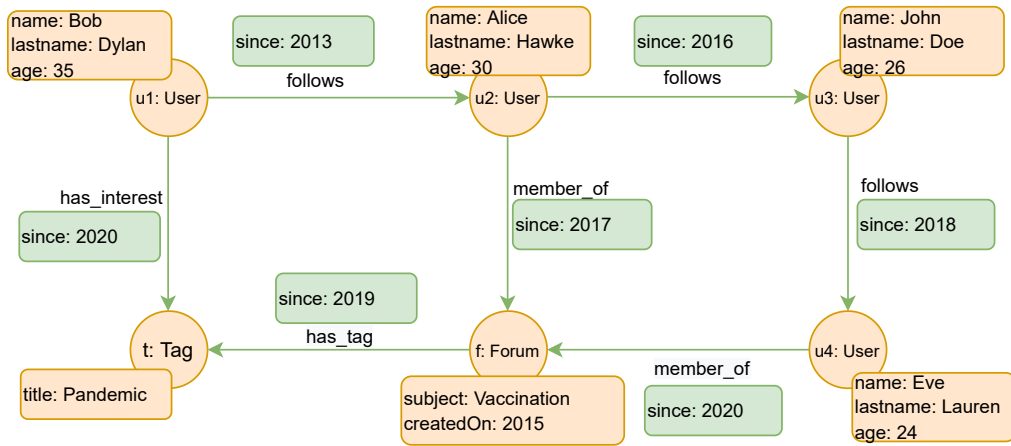
Figure 1.4: Property graph example.

Considering the property graph above a possible representation using tables can be the example shown in Figure 1.5.

User

vid	name	lastname	age	...
101	Bob	Dylan	35	
102	Alice	Hawke	30	
103	John	Doe	26	
...				

Tag

vid	title	...
301	Football	
302	Pandemic	
303	Astronomy	
...		

Forum

vid	subject	createdOn	...
201	Vaccination	2019	
202	Sports	2018	
203	Fashion	2020	
...			

member_of

src	trg	since	...
101	202	2015	
102	203	2019	
103	201	2017	
...			

has_interest

src	trg	since	...
101	302	2020	
102	301	2017	
103	303	2016	
...			

follows

src	trg	since	...
101	103	2015	
101	102	2016	
102	103	2015	
...			

has_tag

src	trg	since	...
201	303	2019	
202	302	2018	
203	301	2020	
...			

Figure 1.5: A possible property graph representation using tables.

1.2   Graph query languages

We first introduce some preliminary notions before we present graph query languages.

1.2.1   Important design choices on query languages

Set semantics returns an answer with the set of bindings that are solutions; *Bag semantics* returns the same answers but allowing multiplicity.

No-repeated-node semantics - The same node cannot occur more than once in the result set of a graph navigation query [Sharma et al., 2021].

No-repeated-edge semantics - Only paths that have distinct edges are allowed. In cases where cycles exist in a graph database, infinite paths are not returned in the evaluation of a graph navigation query [Angles et al., 2017].

As shown in [Sharma et al., 2021], there are two types of evaluation semantics for graph pattern matching:

- *homomorphism based semantics* - all possible sub-graphs that answer the query are returned in the result set. Multiple node and/or edge variables in a graph pattern can match to same node and/or edge in the graph database.

- *isomorphism based semantics* - in this case there are restrictions on the result set based on conditions that either a node variable match to a single node (no-repeated-node semantics) or an edge variable only match to a single edge (no-repeated-edge semantics) or both node and/or edge variable match a single node and/or edge in a graph database (no-repeated-anything semantics) [Angles et al., 2017].

1.2.2   The SPARQL standard for knowledge graphs

SPARQL [Seaborne et al., 2008] is the query language for RDF graphs standardized by the W3C (World Wide Web Consortium).

SPARQL is a declarative query language. For example, the following SPARQL query is written against the RDF graph example shown in Figure 1.2:

<div align="center">

SELECT ?user, ?forum
 WHERE {
         ?user : `follows` Jane .
         ?user : `member_of` ?forum
         }
</div>

Figure 1.6: Simple SPARQL query example.

This query returns the set of pairs consisting in a user and a forum such that the user is a direct follower of Jane and is a member of the given forum.

A typical SPARQL query consists in a SELECT block and a WHERE block. In the SELECT block the variables to be extracted are declared. These variables represent the query results. Variables are written with a ''?'' as a prefix.

The goal of the WHERE block is to define a set of conditions that relate the variables to be extracted by the SELECT block. For this purpose, the WHERE block consists in a set of conjuncts regrouped in a so-called *basic graph pattern* (BGP). Each conjunct is a triple pattern. A triple pattern is essentially an RDF triple where the subject or object can be a variable (whose value is to be extracted).

For example, the triple patterns expressed in the query above are:

<div align="center">

?user: `follows`   Jane
</div>

and

<div align="center">

?user: `member_of`   ?forum.
</div>

Based on the SELECT and WHERE blocks, the above query asks for users that follow Jane directly and are members of a forum. When evaluated on the graph instance shown in Figure 1.2 the result is the pair (`Tim, Forum of Singers`).

In version 1.1 of the SPARQL standard, triple patterns are extended to support property paths. A property path adds the possibility of using regular expressions to match paths of arbitrary lengths. A simple example is shown in the following query:

That query uses the transitive closure `follows+` in order to match sequences of arbitrary length of edges labeled with `follows`. This allows, for example, query of Figure 1.7 to retrieve direct or indirect followers of Jane (not only direct). More generally, SPARQL property paths also allow constructs such as UNION, FILTER and OPTIONAL.

For example, the evaluation of the query of Figure 1.7 on the graph of Figure 1.2 returns the following results (that are all the possible bindings of

```
SELECT ?user, ?forum
 WHERE {
        ?user : follows+ Jane .
        ?user : member_of ?forum
        }
```

Figure 1.7: SPARQL query example with property path.

`?user` and `?forum`):

```
(Tim, Forum of Singers)
(Ryan, Forum of Actors).
```

We see more result tuples are being returned because not only direct followers of Jane are considered, but the indirect ones as well. The graph considered is a really small example, but on a much bigger graph this is much more visible.

1.2.3   Query languages for property graphs

Unlike for RDF graphs there is no standard query language for property graphs (neither standardized by some standardization organization nor de facto standard used in practice). In this section, we review the most advanced query languages for property graphs.

Gremlin   Gremlin [Gremlin, 2022] is a query language developed in the graph computing framework Apache TinkerPop [TinkerPop, 2022]. Gremlin is a functional graph traversal language which is Turing complete. The traversal in the graph is made of steps that can transform the objects, filter or remove them or compute statistics on them. Gremlin traversals can be written in programming languages that support function composition and nesting. Traversals can be written in an imperative manner, a declarative manner, or in a hybrid manner that considers them both. The following Gremlin example illustrates this:

```
G.V().hasLabel('User').has('age', '30')
    .out('member_of').hasLabel('Forum')
```

Figure 1.8: Query example using Gremlin.

In addition to this possible way of expressing graph traversals, Gremlin also offers pattern matching features similar to the ones found in SPARQL. Like SPARQL, it has homomorphism-based bag semantics. Nowadays there exist some variants such as: Gremlin-Java, Gremlin-Python, Gremlin-Scala, Gremlin-Groovy etc.

The query example of Figure 1.8 is written against the property graph shown in Figure 1.4. The evaluation of the query returns all Users of age 30 years old that are members of a Forum. The first part `G.V().hasLabel('User').has('age', '30')` retrieves all nodes with label User and aged 30 years old. The `out` command retrieves all nodes that are reached by the edge labelled '`member_of`' and `hasLabel('Forum')` gets all nodes labelled with `Forum`.
Going a little further on what Gremlin is able to express we give an example that contains the `match` command.

```
G.V().match(
            __.hasLabel('User').has('age', '30')
            .repeat(in('follows').hasLabel('User')).emit().as('x'),
            __.as('x').out('member_of').hasLabel('Forum').as('y')
          )
```

Figure 1.9: Query example using Gremlin.

The example shown in Figure 1.9 uses `match` command to join the two traversals in the graph and to return the values for x and y variables. `repeat` command is used to repeat the operation the amount of times it is needed, with no fixed number of iterations. `as` command declares a variable and '`__`' operator considers that the following operation is applied on the parent traversal one level before.

Cypher Cypher is a declarative graph query language [Francis et al., 2018a]. At first it was developed in Neo4j [Neo4j, 2007], and it has been used in several other products like: SAP HANA Graph, Redis Graph etc. It is widely used in the industry. One reason for its success is that it provides a formal semantics in [Francis et al., 2018b].

In Cypher, a query consists in a sequence of MATCH clauses followed by a RETURN clause. The RETURN clause specifies the values to be extracted. This is accomplished through the use of variables that can match either nodes or edges, and the properties associated with those variables.

Each MATCH clause contains a graph pattern which expresses constraints that must be satisfied by variables.

Variables provide a handy way of naming a node or edge so that it can be referred multiple times in the same query. For example, in the query written in Figure 1.10 the occurrence of the variable "u" in the second MATCH clause refers directly to the variable definition "u" of the first MATCH clause.

Cypher uses a semantics based on no-repeated-edge isomorphism. Once this variable is denoted with all specifications, no adjustments (no further variable affectation) can be made later. As seen in the given query, we can write `follows*`, where `*` is the Kleene star. In Cypher, we can specify the length of the path. Instead of the Kleene star we can specify as well a number such as: 2, 3, 4 etc...

```
MATCH (u:User) -[follows*]-> (x:User {age:"30"})
MATCH (u) -[member_of {since:"2020"}]-> (f:Forum {createdOn:"2015"})
RETURN u.name, f.subject
```

Figure 1.10: Query example using Cypher.

This query returns the name of users that follow directly or indirectly a user aged 30 and the subject of a forum these users are members of.

PGQL   PGQL [van Rest et al., 2016] is a query language for property graphs based on SQL and it is developed by Oracle. It provides a form of pattern matching similar to the one of Cypher. A PGQL query is composed of three main clauses: `SELECT`, `FROM` and `WHERE`. It can also contain optional clauses like `ORDER BY`, `GROUP BY` and `LIMIT`. The result set of a PGQL query is a set of variables and their bindings.

PGQL provides a notation for expressing paths between nodes using regular expressions over sequences of edges. In PGQL we do not only find reachability queries, but also path finding queries. Reachability queries check if a path between pairs exists or not, while path finding queries add the possibility of computing and doing comparisons along the way. In other terms, PQGL allows for encoding graph reachability (transitive closure) queries as well as shortest and cheapest path finding queries.

A PGQL query example is shown in Figure 1.11. In this query, the `FROM` clause accesses the graph information provided under the name `social-network-graph` which refers to the graph shown in Figure 1.4.

```
SELECT u.name
  FROM social-network-graph
 WHERE (u:User) -[follows*]-> (x:User),
       x.age="30"
```

Figure 1.11: PGQL Query example.

This query returns the names of users that follow directly or indirectly a user aged 30.

G-Core   G-Core [Angles et al., 2018] is a graph language designed by LDBC Graph Query Language Task Force whose members come from industry and academia. The intent is to gather the best of both worlds in order to guide the evolution of graph query languages, and to make them more expressive and useful. As of 2022, G-Core is a proposition of a language which is not yet implemented by a system in practice. Two important aspects characterize G-Core and makes

its originality: (i) Composability - the input and output is a graph; (ii) Paths as first-class-citizens.

A query in G-Core starts with a `CONSTRUCT` clause that denotes the result graph to be returned. For example, the query shown in Figure 1.12 is written against the graph `social-network-graph` shown in Figure 1.4. It returns a graph consistsing of nodes (with no edges). The nodes returned are those of Users aged 30 years old along with the properties these nodes have in `social-network-graph`. The `MATCH..ON..WHERE` clause matches graph patterns on a named graph.

```
CONSTRUCT (u)
      MATCH (u:User)
      WHERE social-network-graph
      WHERE u.age='30'
```

Figure 1.12: Query example using G-Core.

The other example of Figure 1.13 shows the particularity of G-Core, the treatment of paths as first-class citizens.

```
CONSTRUCT (u) -@p:followers{distance:=c} /->(m)
    MATCH (u)-/3 SHORTEST p<:follows*> COST c/-> (m)
    WHERE u:USER AND m:USER
      AND u.name='Bob'AND u.lastname='Dylan'
      AND (u)-[:member_of]->()<-[:member_of]-(m)
```

Figure 1.13: Query example with path storage.

In the query illustrated in Figure 1.13, the clause `ON` is omitted because we are considering the default graph `social-network-graph` and this simplifies as we want to focus on paths for this example. Paths are written using this notation '`-/ /-`'. The shortest path is bound by the notation `p<:follows*>` between User '`u`' (for example: User named Bob Dylan) and all Users `m`, under the restriction that both Users are members of the same Forum. In this case only 3 of the shortest paths are returned since this is what is specified in the query. The shortest path cost is bound to variable `c` by the following expression: `p<:follows*> COST c/->`. The cost of the path is its length or hop-count, but it can be defined differently if needed. If the length is not a priority in the result, then `COST c` could be left out. `@p` is a bound path variable of `CONSTRUCT` (u) `-@p:followers{distance:=c} /->(m)` that specifies a stored path. Each one of the paths is saved with the label `:followers` and with the cost as property (`distance` in this example). This query returns a graph that inccludes all nodes and edges involved in these stored paths.

1.2.4   Query language comparisons

We have described the main graph query languages, where they come from, their originalities and their main particularities. From a fundamental perspective, there are basically two interesting dimensions on which they can be compared: semantics and expressivity. Each query language design relies on choices on those two dimensions. We summarize the main differences between the languages due to the choices made in each proposal:

Semantics.

- Cypher is considered as more restrictive because it has a no-repeated-edge based semantics, while PGQL considers a homomorphism based semantics [Sharma et al., 2021, Angles et al., 2017]. In other words, in the result set of a query written in PGQL are found all valid matches even in cases where a variable is used for several mappings of different nodes or edges. SPARQL is built around a homomorphism based semantincs as well.

- An important choice in terms of semantics is the choice made for the return type of the queries. While the majority of languages return sets or sequences of tuples, G-core can return nodes and edges as well, thus a full graph. This is an interesting particularity as it can be unioned with the original graph taken as input.

Expressivity.   When considering expressivity of graph navigation queries, PGQL is considered as more expressive than Cypher because it can consider the Kleene star on path expressions [Angles et al., 2017]. This is made possible by the presence of the `PATH` clause in PGQL. The current version of Cypher (version 9) considers the Kleene star only over edge labels. This limitation is expected to be removed in the next Cypher version (10), not released yet. On the other hand, for graph pattern matching queries, Cypher is considered as more expressive because of the presence of the `UNION` clause.

1.2.5   Recursive query language fragments of particular interest

Several query language fragments for expressing path traversals in graphs have been studied and characterized in the litterature. Their study contributes to the development of the aforementioned fully-fledged practical query languages. We review the main recursive query language fragments and summarize the main results known on them.

Regular Path Queries (over knowledge graphs).   A class of queries which has been well-studied over knowledge graphs is the class of Regular Path Queries (RPQs). RPQs express paths between nodes using a set of regular expression operators. They are one of the most well-studied fragments for querying graphs [Consens and Mendelzon, 1990, Libkin et al., 2016, Barceló et al., 2012, Barceló et al., 2013, Bonifati et al., 2018].

An RPQ is a query of the form $r(x, y)$ where x and y are variables that denote graph nodes, and r is the relation that connects them. In a RPQ, r is a regular expression over edge labels, whose syntax is defined as follows:

$$
\begin{aligned}
r \quad ::= \quad & \\
& v && \text{a single edge label} \\
\mid \quad & r_1/r_2 && \text{concatenation} \\
\mid \quad & r_1|r_2 && \text{alternative} \\
\mid \quad & r^{-1} && \text{reverse} \\
\mid \quad & r^{+} && \text{transitive closure}
\end{aligned}
$$

Notice that this definition of RPQs contains the reverse operator. In the litterature sometimes this fragment is also denoted as 2RPQ to insist on that aspect.

The result of the evaluation of an RPQ $r(x, y)$ is the set of all pairs of vertices x and y that are connected by a sequence of edges that matches the path expressed by r.

For example, the path `a+/b` in the RPQ `?x a+/b ?y` describes all the sequences of edges between `?x` and `?y` that start with one or more successive `a` edges followed by exactly one `b` edge.

CRPQ. Conjunctive regular path queries (CRPQs) are basically extensions of RPQs with conjunction. Specifically, a CRPQ is composed of a head and a body. The head is a non-empty set of vertex variables that are to be extracted, whereas the body is a conjunction of RPQs. An example of a CRPQ is the following:

$$
\begin{aligned}
?y,\ ?z \leftarrow\ & ?x : \texttt{User}\ \texttt{follows+}\ ?y : \texttt{User}, \\
& ?y : \texttt{User}\ \texttt{has\_interest}\ ?z : \texttt{Forum}
\end{aligned}
$$

This CRPQ can be evaluated on the graph of Figure 1.4. It retrieves all users followed directly or indirectly by a certain user (i.e. x in this case), who are members of a Forum.

UCRPQ. Unions of conjunctions of regular path queries (UCRPQs) further extend CRPQs with unions at top level. Specifically, an UCRPQ can be written as $H \leftarrow C_1 \cup ... \cup C_n$ where H is the head that represents the set of variables being extracted and C_i is a CRPQ.

Expressivity of UCRPQs. Notice that the only form of recursion found in RPQs is transitive closure. An important consequence for RPQs (as well as for their CRPQs and UCRPQs extensions) is that they cannot express other forms of recursion like non-regular paths (eg. $a^n b^n$) or the more general ones that compute shortest paths, for example.

1.2.6   Query shapes

Besides language expressivity considerations, studies have identified another important factor that has an impact on the complexity of the graph query evaluation problem. This factor is the "query shape" which refers to the shape of the dependances between the variables in a query.

Specifically, for a given query, the notion of *variable dependency graph* focuses only on the variables that occur in the query, and how these occurrences are connected together using paths (abstracting away from the detail of each path).

Studies in the litterature have identified four shapes that have a direct impact on the computational complexity of the query evaluation problem. We review below each one of these shapes, along with examples.
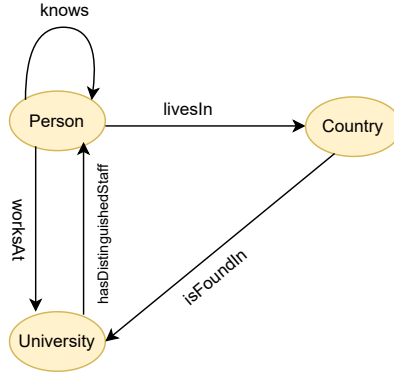
Figure 1.14: Graph example.

1.2.6.1   Linear (chain) queries

Linear queries are queries whose variable dependency graph is a simple path between vertices (i.e. 1–2–3–...–n) where each two consecutive vertices have an edge that connects them (all vertices are of degree 2) except the first and the last one, as illustrated in Figure 1.15.

Figure 1.15: Shape of variable dependency graph of a linear query.

Below we give an example of a query written using UCRPQ and based in the graph shown in Figure 1.14. In this example, the query retrieves the acquaintances of a Person with people who work at a university.

$$?x, ?z \leftarrow ?x : \texttt{Person} \; \texttt{knows} \; ?y : \texttt{Person},$$
$$?y : \texttt{Person} \; \texttt{worksAt} \; ?z : \texttt{University}$$

1.2.6.2   Cyclic queries

Cyclic queries can be seen as linear queries with the same starting and ending node. More precisely, a cyclic query is defined as follows: each two consecutive

vertices in its variable dependency graph are connected by an edge between them. An example of such a shape is shown in Figure 1.16.
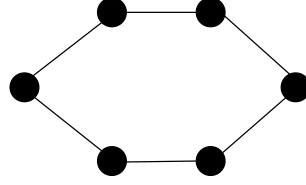
Figure 1.16: Shape of variable dependency graph of a cyclic query.

Below we show an example written in UCRPQ based on the graph illustrated in Figure 1.14. We show the dependences of variables x and y and how they are interrelated. The first conjunct describes a relation between x and y, stating that a person works at a university. The second conjunct describes a reciprocal relation between the same variables, asking whether the person is part of the distinguished staff of that university.

$$?x, \ ?y \leftarrow \ ?x : \texttt{Person worksAt } ?y : \texttt{University},$$
$$?y : \texttt{University hasDistinguishedStaff } ?x : \texttt{Person}$$

1.2.6.3 Star queries

A star query is defined as a query whose variable dependency graph contains a central vertex that is connected to each one of the vertices. An example of this query shape is shown in Figure 1.17.
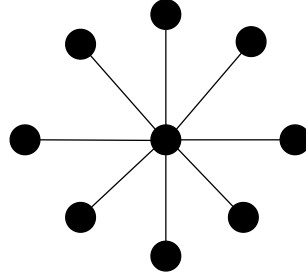
Figure 1.17: Shape of variable dependency graph of a star query.

An example of a query of this shape is written below using UCRPQ and based on the graph shown in Figure 1.14. The central vertex is a person which is connected with each other vertex. These other vertices in this case are a person, an university and a country.

$$?x, \ ?y, \ ?z, \ ?w \leftarrow \ ?x : \texttt{Person knows } ?y : \texttt{Person},$$
$$?x : \texttt{Person worksAt } ?z : \texttt{University},$$
$$?x : \texttt{Person livesIn } ?w : \texttt{Country}$$

The first conjunct describes a relation between two persons, x and y. The second conjunct describes a reciprocal relation between person x and the university he works at, while the third conjunct describes a reciprocal relation between person x and the country he lives in.

1.2.6.4   Clique queries

A clique query is when each two vertices are adjacent in the variable dependency graph. In other words, each two vertices are connected between them by an edge. An example of this query shape is shown in Figure 1.18.
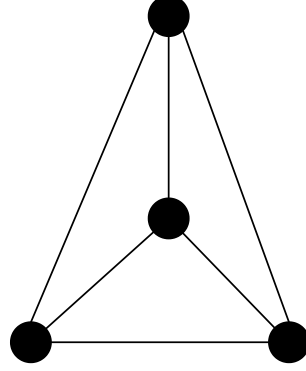
Figure 1.18: Shape of variable dependency graph of a clique query.

An example of a clique query is written below using UCRPQ notation and based on the graph shown in Figure 1.14. Each two variables have a relation between them, making this a clique query and one of the most complicated queries to evaluate.

$$
\begin{aligned}
?x,\ ?y,\ ?z,\ ?w \leftarrow\ & ?x : \texttt{Person}\ \ \texttt{knows}\ \ ?y : \texttt{Person}, \\
& ?x : \texttt{Person}\ \ \texttt{worksAt}\ \ ?z : \texttt{University}, \\
& ?x : \texttt{Person}\ \ \texttt{livesIn}\ \ ?w : \texttt{Country}, \\
& ?y : \texttt{Person}\ \ \texttt{worksAt}\ \ ?z : \texttt{University}, \\
& ?y : \texttt{Person}\ \ \texttt{livesIn}\ \ ?w : \texttt{Country}, \\
& ?z : \texttt{University}\ \ \texttt{isFoundIn}\ \ ?w : \texttt{Country}
\end{aligned}
$$

All these conjucts show relations between different variables that represent persons, universities and countries.

1.3   Complexity of the query evaluation problem

The query evaluation problem   Given a graph database G of size n and a query q of size m the query evaluation problem consists in computing the set of all results of q in G.

In practice $\texttt{m} \ll \texttt{n}$ and this is the reason why there are three main perspectives on the computational complexity of the query evaluation problem:

- *data complexity* where the input is only the graph (the query is considered fixed);

- *query complexity* where the input is only the query;

- *combined complexity* where both the graph and the query are considered as inputs.

As mentioned earlier in Section 1.2.2 basic graph pattern (bgp) is considered to be a set of triple patterns and complex graph patterns (cgp) are bgp augmented with other (relational-like) features like: projection, union, optional, and difference [Angles et al., 2017].

Query complexity has been studied for SPARQL fragments. As discussed in [Angles et al., 2017], the combined complexity when using set semantics and considering complex graph patterns with projection, join, union and filter is NP-complete. It goes to PSPACE-complete When allowing difference and optional operations, it has the same operators as relational algebra, hence the combined complexity is considered to be PSPACE-complete [Vardi, 1982]. As mentioned in [Angles et al., 2017], when considering bag-semantics it remains PSPACE-complete. To the best of our knowledge, the complexity when considering graph query patterns for other query languages for property graphs like the ones we mentioned in Section 1.2.3 has not been studied yet and remains an open question.

The evaluation of RPQs (as defined in Section 1.2.5) is NP-hard in data complexity [Casel and Schmid, 2021] (and therefore in combined complexity). The evaluation of UCRPQs is NL-complete in data complexity and NP-complete in query complexity (and thus combined complexity) [Reutter et al., 2017, Casel and Schmid, 2021].

In practice, it is often useful to identify subclasses of queries that are simpler to evaluate. One of the main results found in [Angles et al., 2017] indicates that, intuitively, the more cyclical the shape of the variable dependency graph (see Section 1.2.6) (i.e., the less it resembles a tree), the more difficult the query is to evaluate.

Given that recursive query evaluation over graphs is a complex problem, it is worth investigating query optimization methods. The goal of those methods is to search for better evaluation strategies for a given query, at the expense of some additional static analysis that can also be complex. This is especially relevant since query optimization methods depend almost only on the query (with few dependence on the graph instance, if any, and anyway $m \ll n$). The next Chapter reviews the main theories and methods developed for optimizing recursive graph queries.

Chapter 2

Foundations of query optimization

2.1 Introduction

In practice, graph instances can be very large with thousands or even millions of nodes or edges. In such graphs, the evaluation of *even relatively simple* queries can become very costly. The evaluation of *recursive* queries is yet much more expensive and difficult. This is because recursive queries may need to process intermediate subquery results whose size can be an order of magnitude bigger than the size of the graph. For this reason, the evaluation of recursive queries can quickly become prohibitively expensive, even on graphs of rather moderate size (depending on their topology). Since recursion brings major benefits for extracting information from graphs (as illustrated in Chapter 1), the optimization of recursive graph queries has attracted a lot of attention lately.

In this Chapter, we review fundamental theories and state-of-the-art methods for query optimization. We start by describing essential theories for optimizing queries without recursion and then methods supporting recursive queries. One of the most fundamental theory for query optimization is Relational Algebra. We describe how it gave birth to different query optimization methods and their specificities. We discuss in particular transformation-based optimization, and the main properties of the different plan enumeration methods and algorithms. We start from the very first attempts of query optimizers creation to the latest propositions. There were a lot of approaches explored with various properties, resulting in some of them being more relevant than others for recursive graph queries. We report on computational complexity of state-of-the-art approaches found in the litterature and we summarize the most relevant results in a graphical representation. We also discuss algebraic and non-algebraic approaches for handling recursion in query optimizers.

2.2 Relational algebra

The relational model dates back to the '70s, introduced by [Codd, 1970]. This refers to a data model where relations are considered as data structures [Abiteboul et al., 1995, Garcia-Molina et al., 2008], and where there exists a relational algebra (RA) in order to specify queries.

The particularities of the relational model are the storage model and the presence of a high-level query language. Data is stored in relational tables. Using relational algebra, operators can be defined and complex queries can be written to transform input relations to one output relation. As mentioned in [Abiteboul et al., 1995], the simplicity that characterizes the relational model has turned it into one of the fundamentals of databases because of two reasons: (i) many theoretical issues that are relevant to other models can be described using this model and (ii) several techniques help to understand other models in a profound way. There exists a standard language that manages data in a RDBMS (Relational Database Management System) called SQL.

2.2.1 Relations

In the relational model a database is a set of relations (tables). A relation is a table composed of rows and columns. Each row contains information about a given object and is called a tuple. Each column has a specific name which is called an attribute (e.g. age). A schema of a relation is the relation name and its set of attributes.

The following example presents the relation Actors. Its schema is `Actors(name, lastname, age, gender)`. For example, one tuple of relation Actors is: (`Blake, Lively, 34, F`) which corresponds to a value for each column attribute.

name	lastname	age	gender
Ryan	Reynolds	45	M
Blake	Lively	34	F
Sandra	Bullock	57	F
Hilary	Duff	34	F
Stephen	Amell	41	M

In the native RA, each component of each tuple should be atomic and of a given type, such as: integer, string etc. This is associated to each attribute of the given relation [Garcia-Molina et al., 2008]. The schema including the type of this description for the relation Actors is as follows: `Actors(name:string, lastname:string, age:integer, gender:string)`.

2.2.2 Data model formalisation

In accordance with the aforementioned explanations, the relation is defined as a set of tuples. It is also classic to see tuples as functions, where for each attribute name a value is returned. This function that takes an attribute name and returns

a value is called a mapping. The relation in this case is defined as a set of mappings. (see Definition 1 and Definition 2 below).

Definitions
We consider:

- $\mathcal{B}$ is a set of values

- $\mathcal{C}$ is a set of column names

- $\mathcal{R}$ is a set of relation names

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

Definition 2: A relation is a finite set of mappings which share the same domain D. The type of the relation is D. Throughout this thesis we will always use this definition. In other terms, when we refer to the *type* of a relation, we refer to the set of attribute names of that relation.

2.2.3 Operations

Operations in relational algebra are used to perform queries. They can manipulate a given relation and create others. Operations can be either unary or binary.
The basic unary operators are:

- **Selection** $\sigma_{\mathfrak{f}}(\varphi)$: when applied to a term φ, the selection $\sigma_{\mathfrak{f}}(\varphi)$ returns the tuples satisfying the filter condition f. Considering the relation Actors given in the example above, f can be a boolean expression such as: `age=34`. The result tuples in this case would be $\{$`(Blake, Lively, 34, F)` , `(Hilary, Duff, 34, F)`$\}$.

- **Projection** $\pi_\theta(\varphi)$: θ is a list of attributes that can be separated using comma. $\pi_\theta(\varphi)$ is applied to relation φ which must have θ as attributes. θ are a subset of attributes found in relation φ. From the totality of attributes of relation φ all are removed except those in θ. Considering the example above, if we want to project only on names and lastnames of the relation Actors we must write the following expression: $\pi_{\texttt{name, lastname}}(\texttt{Actor})$.

- **Renaming** $\rho_a^b(\varphi)$: corresponds to the relation φ for which the attribute a has been renamed into b. The attributes of φ must contain the attribute a but not b.

The basic binary operators are:

- **Union** $\varphi_1 \cup \varphi_2$: simply corresponds to the union of the set of solutions of φ_1 with those of φ_2. Union is only defined for terms that share the same type (set of attributes).

- **Natural join** $\varphi_1 \bowtie \varphi_2$: The natural join is a specific case of join which combines two relations based on all their common attributes. It can be seen as the cartesian product of the two relations, followed by a filter that eliminates tuples in which common columns have different values.

- **Antijoin** $\triangleright$ The relation $\varphi_1 \triangleright \varphi_2$ is the result calculated by removing all the rows joined between φ_1 and φ_2 from the rows of φ_1.

2.2.4   Rewrite rules

One major interest of relational algebra, is that once a term is written in relational algebra, it can be rewritten into semantically equivalent terms. This rewriting is useful to reorder operations and potentially find equivalent terms that can be evaluated more efficiently.

Such a rewriting process is done using rewrite rules. For example, let's consider two relations `Actor` and `Movie`. Then, let's join them and filter the age of Actors. One possibility will be the following: $\sigma_{\texttt{age=34}}(\texttt{Actor} \bowtie \texttt{Movie})$. This means that the join between two tables is computed first and then the selection. In this case, it would be more efficient to apply the selection first (hence the size of the table is reduced) and then compute the join between them. The query is as follows: $(\sigma_{\texttt{age=34}}(\texttt{Actor}) \bowtie \texttt{Movie})$. Both these two queries are equivalent because they compute the same set of solutions, but one is more efficient than the other. Term rewriting by the application of the set of rewrite rules is considered an optimization strategy. Below we show a list of rewrite rules. They are written in the form $t \to t'$, where t is the first term and t' is the term created after the rewrite rule is applied.

We use an auxiliary function $\texttt{filt}(f)$ that takes some filter f as input and returns the set of columns used in f.

2.2.4.1   Rewrite rules for Join

$$\varphi \bowtie (\psi \bowtie \chi) \to (\varphi \bowtie \psi) \bowtie \chi \quad [\texttt{Left Associativity of Joins}]$$

$$\varphi \bowtie \psi \to \psi \bowtie \varphi \quad [\texttt{Commutativity of Joins}]$$

$$\varphi \bowtie (\psi \cup \gamma) \to (\varphi \bowtie \psi) \cup (\varphi \bowtie \gamma) \quad [\texttt{Distributivity of Join over Union}]$$

As shown in [Pellenkoft et al., 1997] for a rule set where the commutativity of joins is considered along with the left associativity, the right commutativity is redundant because it can be deducted by the first two. This is why we do not include the right associativity rule in this list.

2.2.4.2 Rewrite rules for Union

$$\varphi \cup (\psi \cup \chi) \rightarrow (\varphi \cup \psi) \cup \chi \quad [\texttt{Left Associativity of Unions}]$$

$$\varphi \cup \psi \rightarrow \psi \cup \varphi \quad [\texttt{Commutativity of Unions}]$$

2.2.4.3 Rewrite rules for Selection

$$\frac{\texttt{filt}(f) \subseteq type(\varphi) \wedge \texttt{filt}(f) \nsubseteq type(\psi)}{\sigma_f(\varphi \bowtie \psi) \rightarrow \sigma_f(\varphi) \bowtie \psi} \quad [\texttt{Push Selection in Join}]$$

$$\frac{\texttt{filt}(f) \subseteq type(\varphi) \wedge \texttt{filt}(f) \subseteq type(\psi)}{\sigma_f(\varphi \bowtie \psi) \rightarrow \sigma_f(\varphi) \bowtie \sigma_f(\psi)} \quad [\texttt{Push Selection in Join}]$$

$$\sigma_f(\varphi \cup \psi) \rightarrow \sigma_f(\varphi) \cup \sigma_f(\psi) \quad [\texttt{Push Selection in Union}]$$

$$\sigma_a(\sigma_b(\varphi)) \rightarrow \sigma_b(\sigma_a(\varphi)) \quad [\texttt{Commutativity of Selections}]$$

2.2.4.4 Rewrite rules for Projection

$$\frac{type(\varphi) \cap type(\psi) \subseteq S}{\pi_S(\varphi \bowtie \psi) \rightarrow \pi_{S \cap type(\varphi)}(\varphi) \bowtie \pi_{S \cap type(\psi)}(\psi)} \quad [\texttt{Push Projection in Join}]$$

$$\frac{type(\varphi) = type(\psi)}{\pi_S(\varphi \cup \psi) \rightarrow \pi_S(\varphi) \cup \pi_S(\psi)} \quad [\texttt{Push Projection in Union}]$$

$$\pi_{S_1}(\pi_{S_2}(\varphi)) \rightarrow \pi_{S_2}(\pi_{S_1}(\varphi)) \quad [\texttt{Commutativity of Projections}]$$

As an important purely logical optimization strategy, notice that pushing selections and projections always improve performance, since these two operations reduce the size of intermediate results, so the earlier (closer to the source) the better. For the other rewrite rules the situation changes. Some examples where the performance improvement is unclear and is dependant on the sizes of relations of each example might be the case of changing the order of join operations, or the distributivity of joins over unions. For this reason, query optimizers also need to rely on other metrics to assess whether the rewriting is actually an optimisation, based on e.g. data cardinality estimations.

2.3 Query optimization

Query optimization is a complex phase because many aspects must be taken into account. Many choices must be made. For instance, the considered set of rewrite rules might be more or less rich; how to apply rewrite rules to minimize redundant computations, etc.

In this section we focus on the different plan enumeration techniques (i.e. how to properly apply a set of rewrite rules) found in the litterature. We explain important considerations during optimization phase and what are the current approaches used in modern query optimizers. For this purpose, we start by defining some important concepts, such as the plan space.

2.3.1   Plan space

For a given term t and a given set R of rewrite rules the *plan space* is the set of all terms semantically equivalent to t that can be obtained by the application of an arbitrary sequence of rewrite rules in R.

An ideal optimization method relies on generating the plan space exhaustively, however this is often made very complicated in practice due to heavy combinatorial issues. Plan spaces are usually very large. For complex queries, we might need to apply some guidance during plan exploration. Various forms of heuristics have been proposed in the litterature.

One of most known heuristic is to explore only left-deep plans[1], which is implemented in one of the first query optimizers [Selinger et al., 1979]. Then there can be other ways of searching like: right-deep, zig-zag etc. The idea is to have some sort of restriction of the plan space when generating equivalent plans, so as to lower the burden. Also, another type of guidance has been proposed in the so-called *cost-based optimizers*. The idea is to direct the plan space exploration based on the cheapest plan found at a given moment. For example, when enumerating the plan space, the search can be directed in a certain way such that it always follows the direction of the less estimated expensive plan.

The goal when generating the plan space is to be as efficient as possible and to avoid redundant calculations. To achieve this not only we can follow the aforementioned types of guidances, but we can also rely on the analysis of the sequence of transformation rules as well. This has been studied in [Pellenkoft et al., 1997], where the history of the application of rewrite rules is somehow recorded in order to retain which rules have already been applied and which ones are still useful to apply. This way one can avoid generating duplicates, to a certain extent.

2.3.2   Properties of plan enumeration algorithms

We identify three interesting properties for an algorithm that generates plan spaces:

- **Correctness:** the generated plan space must only have semantically equivalent plans. The correctness is ensured by the individual rewrite rule application. For each one of the rewrite rules, we make sure that when applied, the semantics of the term is preserved.

- **Completeness:** The ability to enumerate the whole plan space possible when considering an initial query q and a given set of rewrite rules r is called completeness. This property ensures that every rewrite rule is triggered

---

[1]i.e. omit the exploration of bushy plans for instance.

when necessary for completely expanding the plan space when possible. This property is proved in [Roy, 2001] for an algorithm that applies rewrite rules on select-project-join (SPJ) queries. The proof is done by contradiction: there cannot exist a plan created by the application of a given set of rewrite rules r in a given query q that is not generated by the algorithm of rewrite rule application.

- **Efficiency:** plan enumeration phase can become very tricky (complex) because of its high computational complexity. Plan enumeration is costly in space and time (we will review complexities in Section 2.3.5).
  For different query shapes (See Chapter 1), the complexity varies. The worst cases are known to be the cases of star and clique queries [Ono and Lohman, 1990]. Star queries have a higher practical importance since they are really common (e.g. UCRPQs), while cliques do not have much practical value. Also some algorithms can be better for certain query topologies than others. As shown in [Moerkotte and Neumann, 2006], one of the algorithms `DPsize` is superior to `DPsub` for chain and cycle queries, while `DPsub` is better when considering star and clique queries.

2.3.3 Algorithmic Techniques for Plan Enumeration

Plan enumeration phase is one of the crucial steps of a query optimizer. There are several ways to enumerate plans in query optimization phase. The approaches used in traditional query optimizers are bottom-up [Selinger et al., 1979], top-down [Graefe and McKenna, 1993, Graefe, 1995] and randomized algorithms [Steinbrunn et al., 1997]. In Figure 2.1 we show these sub-groups, which will be detailed further.
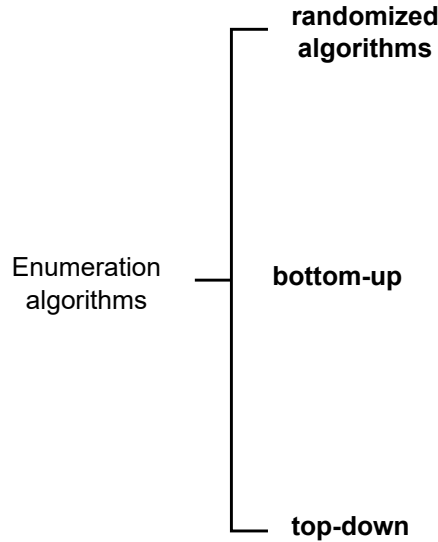
Figure 2.1: Enumeration algorithms.

- In the case of randomized algorithms there are two algorithms: Iterative Improvement (II) and Simulated Annealing (SA) or even Two Phase Opti-

misation (2PO) that considers them both. II is an algorithm that starts at a random state, goes for the local optimization by accepting random downhill moves until it reaches a local minimum. This is repeated until the stopping condition is met. Moving through to SA, which in difference with II accepts uphill moves using probability, but avoiding a high cost local minimum. Then there is the 2PO which is a combination of both of them. Phase one follows II idead, where some local minimums are found, and then is the other phase that takes charge.

- Bottom-up approaches generate the plan space by starting from the leaves (initial relations) and going up in the tree of operators when progressively exploring alternate combinations of operators. Plan enumeration phase is as follows: Each relation is completely optimized in the moment it is accessed. For example: if we need to optimize A ⋈ B ⋈ C, the optimization will be in the order where first [A], [B], [C] will be fully optimized, then is [AB], [AC], [BC] and in the end it is [ABC].

- Top-down approaches start from the root and recursively explore sub-branches in search for possible alternatives. In the top-down approach the optimization is done in a different order and using a different structure. It is structured in a particular way to group all semantically equivalent terms. It starts from the initial query and continues optimizing the subgroups in order. If we refer to the previous example of the bottom-up approach the inital term is [ABC] and then it optimises the other smaller subgroups.

2.3.4   Join Enumeration
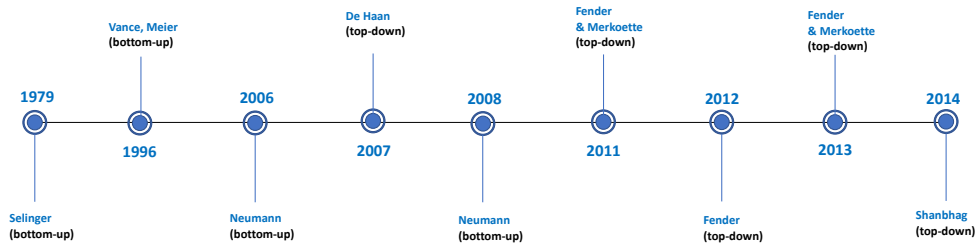
Figure 2.2: Timeline of works on join order optimization for bottom-up and top-down approaches.

The most studied queries in the litterature are select-project-join (SPJ) queries. In both bottom-up and top-down approaches, the main focus goes on join enumeration because of its high processing cost. Join ordering has been proven to be a NP-hard problem [Ibaraki and Kameda, 1984]. There have been several studies throughout the years to optimize join ordering. In Figure 2.2 there can be found some of these works conducted through the years. We focus on works that consider bottom-up and top-down approaches.

Joins are referred as the backbone of the query optimization in [Neumann and Radke, 2018], thus highlighting its importance. Queries considered almost anywhere in litterature are the ones containing selections, projections and joins. Since selection and projection do not add in the combinatorics, they are "ignored" and the main attention is put into join enumeration because of its complexity. More details on how a separate phase of join enumeration is considered can be found in Section 2.3.6.

Further we discuss for the works conducted divided into two big subgroups, bottom-up and top-down.

In bottom-up approaches we start with the pioneer in this field [Selinger et al., 1979] that proposes a dynamic programming algorithm for join ordering by generating the optimal partial plans in the order of increasing size. They limit the search space by considering only left-deep trees which speeds up the optimization. However, bushy plans are often more efficient in practice. In [Vance and Maier, 1996] they propose an algorithm considering different partial table sets that not only search the left-deep plans, but also includes cartesian products to create optimal bushy join trees. Both these approaches apply operations to checkif subgraphs are connected. By these two, the lower bound in [Ono and Lohman, 1990] was not achieved. The work found in [Moerkotte and Neumann, 2006] proposes a graph based dynamic programming algorithm that generates the connected subgraph and organizes the search by the graph structure. This way, it does not need to calculate the combinations and connections that are not part of the final result. Hence it can perform in a more efficient way. Some improvement of the previous algorithm is made in [Moerkotte and Neumann, 2008] by the same authors by including non-inner joins and generalizing it for hyper graphs.

In top-down approaches in [Pellenkoft et al., 1997], they propose a way to lower the complexity of the enumeration process and achieve the lower bound of $O(3^n)$ given by [Ono and Lohman, 1990] by avoiding the generation of duplicates. This approach considers cross products. To reach this, they enable or disable the transformation rules based on their usage. For example, considering the commutativity rule of joins, there is no need to re-apply it after being triggered once, because it will generate the original term.

In [DeHaan and Tompa, 2007] a top-down join enumeration algorithm using the idea of the minimal cuts to generate the connected subgraphs is proposed. They avoid exhaustive enumeration by using the strategy of branch-and-bound. Hence, the algorithm performs faster. This method shows that the top-down methods can be efficient as well. The research continues with the algorithm proposed in [Fender and Moerkotte, 2011] which is easier to implement than the algorithm

proposed in [DeHaan and Tompa, 2007]. Also they consider all possible bushy join trees, but cut out all cross products from the search. The following year, the same authors proposed another top-down method in [Fender et al., 2012] which has better runtime performance and with improved pruning techniques. Then in [Fender and Moerkotte, 2013b, Fender and Moerkotte, 2013a] the same authors continue on the same path by presenting an algorithm that handles non-inner joins and is made for hyper graphs as well. An interesting algorithm based on [Fender et al., 2012] is proposed in [Shanbhag and Sudarshan, 2014], which enumerates cross-product free trees of join operators in a complete manner.

To conclude, similar results are achieved from both these approaches, making them competitive between each-other. As we can observe in Figure 2.2 research at first was mainly focused on bottom-up approaches, until [DeHaan and Tompa, 2007] showed that similar works for top-down approaches can be efficient as well. Then, the focus was shifted towards studying join order optimization for top-down approaches.

2.3.5   Computational complexity of plan enumeration

One of the biggest issues of query optimizers are their costs in terms of resources: memory and computation. During the plan enumeration phase we should find and gather all the possible plans to be able to find the optimal one in the end. With the application of rewrite rules the amount of plans keeps growing until it reaches a combinatorics explosion. In this case it is especially relevant to study the computational complexity, and in this case lower complexity bounds in particular. Defining lower bounds gives an insight on the expected behavior of a query optimizer. There have been a lot of studies for space and time complexity, defining upper and lower bounds for the bottom-up or top down approaches; general case (exhaustive search) or when considering just left-deep plans, with or without bushy plans or cross products; even when considering different queries like linear, star or clique. One of the earliest mentions for the complexity is in [Ono and Lohman, 1990], where it is reported on time complexity for the bottom-up approach with cartesian products. Complexities are expressed in terms of the query size n which implicity corresponds to the number of joins in SPJ queries. Complexities are as follows: $O(3^n)$ (with bushy trees) and $O(n*2^n)$ (without bushy trees). Because of the cartesian products, the shape of the query does not change the complexity.

When removing the cartesian products, the shape of the query matters. They report on the worst case for linear queries (with bushy trees) which is $O(n^3)$, in contrast to $O(n^2)$ (without bushy trees). The complexity of star queries is $O(n*2^n)$ and the presence of bushy trees does not change the complexity in this case, since the form is a "hub", and the center is part of all joins.

In [DeHaan and Tompa, 2007] a lower bound is reported for space complexity for top-down approaches $\Omega(n*2^n)$ when considering left-deep plans and $\Omega(3^n)$ when including bushy plans. For bottom-up approaches they report on a lower bound of $\Omega(2^n)$ for left-deep/bushy plans storage.

In Figure 2.3 we have presented some of the results considering time complexity.

There we have illustrated the complete picture of where the bottom-up approaches stand. We have shown the results of the most promising approaches proposed in the litterature when considering SPJ queries. There we indicate the complexity of three approaches presented in [Moerkotte and Neumann, 2006], where it is shown that for different query topologies, different approaches must be chosen. For example, DPsize algorithm [Moerkotte and Neumann, 2006] performs better for chain(linear) and cycle queries, whereas DPsub algorithm [Moerkotte and Neumann, 2006] performs better for star and clique ones. A new approach is presented in the same paper with more promising results, especially for star queries which are very much used in a lot of data warehouses. For top-down approaches one of the most advanced is RS-Graph [Shanbhag and Sudarshan, 2014], a Volcano based approach with cross-product suppression, but which includes bushy plans. The worst-case complexity is for clique queries and is calculated as $O(3^n)$.

To conclude, as shown in Figure 2.3 the bottom-up and top-down approaches match for the worst-case time complexity considering the query topology (clique queries). The studies for complexity are conducted for SPJ queries, considering the complexity of join enumeration.
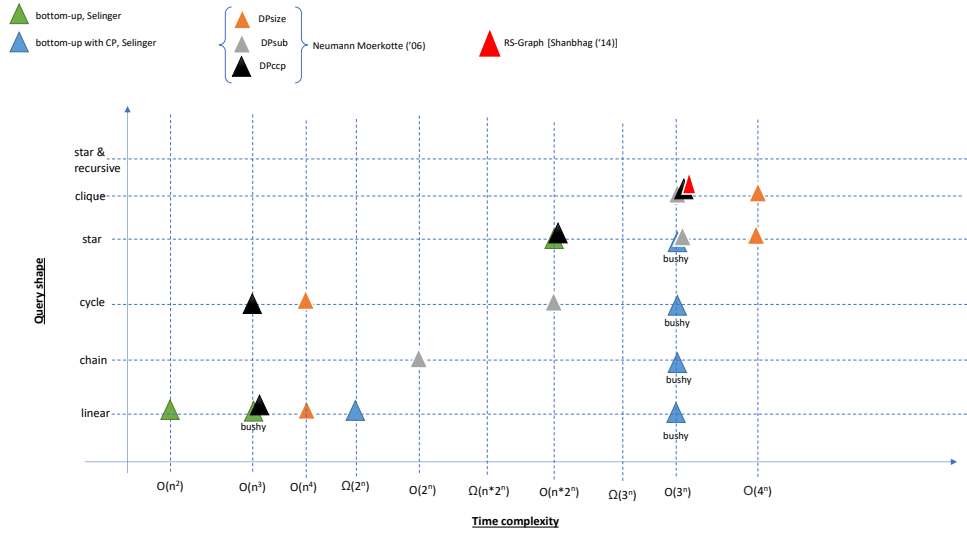
Figure 2.3: Complexity of bottom-up and top-down approaches.

2.3.6   Union in transformation-based query optimizers

Principles and limits of SPJ optimization techniques   Most of the works on query optimization focus on select-project-join (SPJ) queries. In order to report on the optimization of SPJ queries we introduce a separation in 3 stages:

$Stage_1$ refers to the phase of pulling selection and projection operators in order to reach a stratified form where selections and projections only occur at top level (and joins remain underneath). In order to reach that stratification, some

rewrite rules to pull those operators are used (specifically they correspond to the reciprocal rules of Pushing selection in Join or Pushing Projection in Join found in Section 2.2.4). Pulling Selection from Join is as follows: $\varphi \bowtie \sigma_\theta(\psi) \to \sigma_\theta(\varphi \bowtie \psi)$. The same works for projections: $\pi_a(\varphi) \bowtie \psi \to \pi_a(\varphi \bowtie \psi)$. Figure 2.4b illustrates the stratified form reached after $Stage_1$.

$Stage_2$ is the optimization phase of join enumeration, which is the topic of most approaches found in the litterature. This is the part shown in the red square in Figure 2.4b.

$Stage_3$ Lastly, once the most efficient join ordering is found after the join enumeration phase, selections and projections are pushed back to the bottom of the optimized form. This is done using the rules of pushing selections and projections as in Section 2.2.4. This stage is called $Stage_3$ and an illustration of the form reached after this stage can be found in Figure 2.4c.

The way to optimize SPJ queries is to apply all these stages, in order, one after the other: first the pulling of $Stage_1$, then the join enumeration of $Stage_2$, and finally $Stage_3$. One example is illustrated in Figure 2.4 for each stage application.

Most works found in the litterature address $Stage_2$ because it is, by far, the most complex one. This is because $Stage_1$ and $Stage_3$ only consist in a linear traversal of the tree of operators, that do not add any combinatorics. This makes them negligible in front of the complexity of join enumeration which is exponential, and widely known to trigger explosive combinatorics in practice TODO:cite the paper a partir de 10 joins on oublie.

Union in query optimization. Unions have been largely neglected in these studies, they are usually simply not taken into account at all as pointed out by [Chaudhuri, 1998]. It is unclear how the complexity studied for SPJ can be transferred or not for SPJU (queries in which union can appear anywhere as well; and not only at top-level). If we add unions in a given query the combinatorics grow just by triggering the rewrite rule of distributivity of join over union $\varphi \bowtie (\psi \cup \gamma) \to (\varphi \bowtie \psi) \cup (\varphi \bowtie \gamma)$. As shown in this rule, new terms being joined are created. In these terms, selections and projections are present as well. More importantly in this case we cannot apply a $Stage_1$ for pulling selections and projections. The reason is that the classic type system is too restrictive. The typing allowed for union is as follows:

$$\frac{\Gamma \vdash \varphi_1 : \tau \qquad \Gamma \vdash \varphi_2 : \tau}{\Gamma \vdash \varphi_1 \cup \varphi_2 : \tau}$$

Given a schema Γ, the typing judgment $\Gamma \vdash \varphi : \tau$ means that when evaluated in an environment conforming the schema Γ, φ yields a relation of *type* τ. For the union to be allowed, both terms φ_1 and φ_2 must be of the same *type* τ. In this case, the type system of union is too strict to allow for $Stage_1$. Let's consider an example where a rewrite rule considering selection and union is applied. When applying the rewrite rule of pulling selections on a term $t = \varphi \cup \pi_a(\psi)$, a new

term $t' = \pi_a(\varphi \cup \psi)$ will be created. When selection is pulled from t, the type of the second operand of the union changes (the new type contains column a). When the strict type system for union is considered, the creation of t' is not allowed since the two operands of the union in t' have different types, which violates the above typing judgment.

In order to be able to reuse results from the litterature on SPJ queries, one would first need to transform a term into a stratified form (like in Figure 2.4b) where only join operators occur in subtrees so as to apply $Stage_2$ on them. However, the presence of unions makes this complicated: when unions are present, it is not clear how to overcome the aforementioned barrier since a join enumeration optimization phase ($Stage_2$) would not apply directly anymore.

We see two possible approaches to overcome this problem: (i) Continuing with the strict type system and as a consequence do not allow a first phase of pulling and a separate optimization considering only joins, or (ii) Disabling the strict type system and allow rewrite rules to generate unions in which operands can have different type for the purpose of pulling the projections. Each approach has its advantages and drawbacks and we comment on them:

- In the first approach, an optimization phase considering only join enumeration cannot happen because of the strict type system which would block a first phase of pulling for certain operators. This basically means that SPJ approaches from the litterature are hardly reusable, if resuable at all. They would need to be revisited and probably heavily modified due to the presence of unions.

- In the second approach, one possible way to enable a separate phase for join enumeration is to adapt the type system so as to be able to perform $Stage_1$ (pull all the other operators except join). A separate phase for join enumeration for SPJ could then happen: this would allow to reuse algorithms from the litterature with interesting complexity bounds for this stage. However, this would require investigating techniques to be able to track the projected columns to push them back in a correct $Stage_3$ that preserves semantics.

In practice, recent cost-based query optimizers [Begoli et al., 2018, Soliman et al., 2014] support many operators, including union. For instance [Begoli et al., 2018] and [Soliman et al., 2014] do not leverage known join enumeration optimization techniques and trigger rewrite rules directly when possible. As a consequence, they do not leverage the best complexity results for join enumeration known in the litterature (they cannot for the aforementioned reasons).

In conclusion, there are many papers studying SPJ queries that provide enumeration techniques with interesting complexity bounds. However to the best of our knowledge, the complexity of plan enumeration with SPJU queries has not been studied so far. In practice, optimization techniques for SPJU queries have been implemented, but the complexity of plan enumeration is unclear since it cannot be derived directly from the techniques developed for SPJ queries.

(a) Inital term.

(b) Term after $Stage_1$, ready for join enumeration.

(c) A sample term obtained after join enumeration and $Stage_3$.
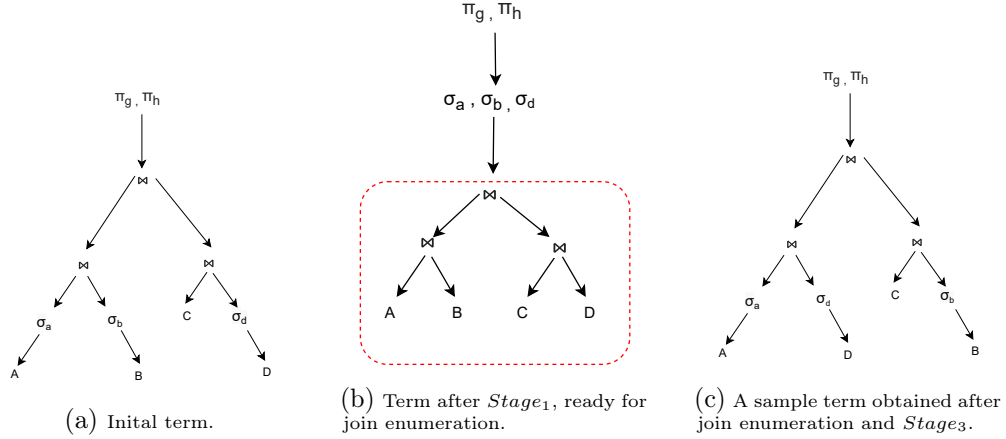
Figure 2.4: Example of a simple query.

2.4 Implementations of transformation-based optimizers

In this section we review the main system implementations of the fundamental studies on transformation-based optimizations found in the litterature. The first part is focused on implementation of bottom-up approaches, whereas the second part is focused on implementations of top-down approaches.

2.4.1   Background on bottom-up approaches

2.4.1.1   System R

System R [Selinger et al., 1979] is a database management system that is based on the relational model of data. The SQL statement passes through 4 phases of processing: Parsing, Optimization, Code Generation and Execution.

Plan of query optimization   Relations are stored in the storage system as a collection of tuples with physically contiguous columns. Tuples are stored in pages; pages are organised in segments (logical units). The way to acces a tuple in a relation is from scans, which are of two types: segment scan and index scan. Segment scan finds all the tuples of the given relation. All pages that contain tuples are examined and the tuples belonging to the given relation are returned. Index scan: The index is created by the System R user on one or more columns of a relation. Indexes and relation tuples are saved on separate pages. These indexes are implemented as B-trees, with leaves that are pages which contain keys and identifiers of tuples that contain that key.

Enumeration of Alternative Plans   There are two different types:

- Single-relation plans

- Multiple-relation plans

Queries over a single relation are composed of select, project and aggregate. Access paths with the lowest cost are chosen. Different operations are made together, like for example when an index is used for a selection, projection is performed for each retrieved tuple, and all these tuples are pipelined into the aggregate computation.

Whereas for queries over multiple relations, a fundamental decision is done in System R where only left-deep join trees are considered. Search space pruning becomes a necessity as the number of joins grows very fast. Left-deep plans differ only in the order of relations, the access method for each relation and the join method for each join. For each subset of relations, only the cheapest plans are retained. A join operator is chosen over a Cartesian product. Left-deep trees are preferred by the system because they generate almost all fully pipelined plans. Intermediate results are not written to temporary files. All single-relation plans are first enumerated (selections and projections are taken into account as early as possible) and then other operators like Order By, Group By, aggregates etc. are handled as a final step.

2.4.1.2 Starburst

Query optimization in the Starburst project [Haas et al., 1989] starts with a structural representation of the SQL query that is used during optimization, called the Query Graph Model (QGM). In the QGM, a box represents a query block and labeled arcs between boxes represent table references across blocks. Each box contains information on the predicate structure as well as on whether the data stream is ordered.

In the query rewrite phase of optimization [Pirahesh et al., 1992], rules are used to transform a QGM into another equivalent QGM. Transformation rules are applied only after being checked on conditions of applicability. Rules may be grouped in rule classes and it is possible to tune the order of evaluation of rule classes to focus search. Since any application of a rule results in a valid QGM, any set of rule applications guarantee query equivalence (assuming rules themselves are valid). The query rewrite phase does not have the cost information available, so it either retains alternatives obtained through rule application or it applies a heuristic way (by compromising optimality).

Then, in the plan optimization phase, given a QGM, an execution plan (operator tree) is chosen. In Starburst, the physical operators (called LOLEPOPs) may be combined in a variety of ways to implement higher level operators and they are expressed in a grammar production-like language [Lohman, 1988]. While doing these combinations, comparable plans that represent the same physical and logical properties but have higher costs, are pruned. Each plan has a relational description that corresponds to the algebraic expression it represents, an estimated cost, and physical properties. These properties are propagated as plans are built bottom-up. Thus, with each physical operator, a function is associated that shows the effect of the physical operator on each of the above properties.

2.4.2 Background on top-down approaches

2.4.2.1 Volcano/Cascades

Given a specific logical algebra, physical algebra and rules, Volcano [Graefe and McKenna, 1993] generates a top-down query optimizer. Volcano integrates both logical and physical steps into a single top-down application of transformations. When the DBMS is operational and a query is entered, the query is passed to the optimizer, which generates an optimized plan for it. It starts with one logical expression and apply transformation rules to obtain new expressions. It keeps the cheapest physical expression that implements a logical expression and physical properties. The output of the optimizer is a plan, which is a physical expression.

Logical Space Generation The data structure used is Logical Query Directed Acyclic Graph (LQDAG), which we will describe more in detail in Section 2.5. The algorithm for plan space generation expands LQDAG by applying all possible transformation rules.

The enumeration phase is handled in a top-down, depth-first search using recursive algorithm. It uses a memoization structure to cache best plans for each equivalence node for future re-use and prunes choices whenever they exceed cost limit.

Cascades [Graefe, 1995] is an extension of Volcano, where one of the novelties is that it allows to use operators that are both logical and physical for predicates. In Cascades search strategy some guidance is applied to rule sets. In the cases where guidance cannot be used, the Volcano search strategy is applied instead. Some recent query optimizers that are based on Volcano/Cascades strategy are Orca [Soliman et al., 2014] and Calcite [Begoli et al., 2018].

2.5 Zoom on the LQDAG approach

In this section we present in more details the Logical Query DAG (LQDAG) approach [Graefe and McKenna, 1993] because this is a main building block used in the contributions of this thesis. The LQDAG is a directed acyclic graph data structure used to represent and generate the logical plan space. This representation allows the sharing of common subparts. It was introduced in [Graefe and McKenna, 1993] and improved in [Graefe, 1995] by the same authors. It is also well described as the AND-OR-DAG in [Roy et al., 2000, Shanbhag and Sudarshan, 2014] where it is used for detecting and unifying common subexpressions for multi-query optimization [Roy et al., 2000]; and for generating the space of cross-product free join trees [Shanbhag and Sudarshan, 2014].

Definition of data structure The LQDAG contains nodes of two different types: *equivalence* nodes and *operation* nodes. Equivalence nodes can only
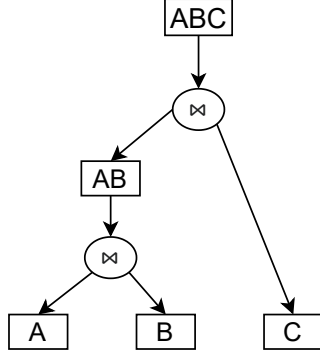
Figure 2.5: An initial LQDAG.
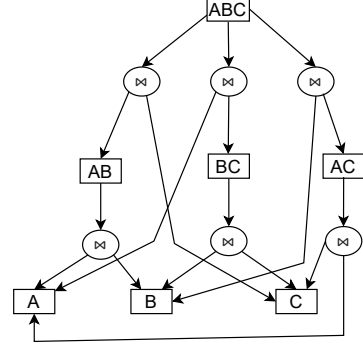
Figure 2.6: An expanded LQDAG.

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 [Roy et al., 2000], Figure 2.5 illustrates a sample LQDAG, and Figure 2.6 depicts its expansion obtained after the application of commutativity and associativity rules for the join operator.

2.6 Recursion

As illustrated in Chapter 1, recursion in graph queries is very powerful and essential as it allows one to extract information from direct and indirect connections in the graph. Over the years, several works have attempted to optimize recursive queries over graphs. In this section, we review the main approaches and classify them into two main sub-groups: Non-relational and relational algebra approaches.

2.6.1 Approaches not based on relational algebra

2.6.1.1 Datalog.

Datalog is a declarative logic programming language based on Prolog and is focused on data. It is well-known as a language designed for recursive queries.

The Datalog line of works [Alvaro et al., 2010, Fan et al., 2019, Francis-Landau et al., 2020, Huang et al., 2011, Seo et al., 2015, Shkapsky et al., 2016, dat, 2021, Wang et al., 2015] developed methods for optimizing recursive queries formulated in Datalog: magic-sets [Bancilhon et al., 1985, Saccà and Zaniolo, 1985, Gardarin, 1987], demand transformations [Tekle and Liu, 2011], automated reversals [Naughton et al., 1989], and the FGH rule [Wang et al., 2022]. Datalog has some stratification restrictions when using negation and recursion.

A Datalog program is composed of facts and rules. An example of a fact is considered $A(X, Y)$ and it can be read as: Y is in relation A with X.

The rule itself is composed by the head and the body. The head is on the left

side, and the body is on the right side. A variable of the left side should always be found in the right side. An example of a rule can be as following:

`ancestor(X, Y) :- parent(X, Y)`

`ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).` This can be read as: X is an ancestor of Y if X is a parent of Y. X is an ancestor of Y if X is a parent of some Z, and Z is an ancestor of Y.

In Datalog, only the loop body is optimized, not the whole loop. One system that does that is Souflée, but limited optimization techniques are applied (magic-sets and semi-naive evaluation) and are mainly restricted to positive Datalog queries. Below, we list optimization techniques in Datalog:

(1) Magic Sets [Bancilhon et al., 1985, Saccà and Zaniolo, 1985] - are used as a technique to reduce the number of not used tuples. These are removed by considering the constraints. For a given query, the datalog program can detect a certain behavior and will avoid calculating several times the same one.

(2) Right and left linear programs - For a transitive closure $R+$ there can be two different translations, from the left or from the right. As shown in [Naughton et al., 1989], for a Datalog term that computes a binary relation as A (x, y), when in right-linear program the filter can be pushed on the right side (y in this case) and vice versa, on the left side (x in this case) when in left-linear program.

(3) Demand transformation [Tekle and Liu, 2011] - is an improvement over the Magic Sets. It uses the same idea as the improvements of Magic Sets: it pushes filters to not compute useless facts. Even though it performs better than Magic Sets, being based on them it has the same issues, for example: sensitivity to programs that are right or left.

(4) The optimization framework proposed in [Wang et al., 2022] gathers magic-sets, semi-naive evaluation and proposes a new FGH rule for optimizing recursive Datalog programs with aggregations. They focus on a framework that uses methods from program synthesis. They are able to express the other techniques as: magic-sets and semi-naive evaluation and some new optimizations. FGH-Rule optimization allows directly the computation of the connected component label of every node x as the minimum of its own label and the smallest component of its neighbors by using a single recursive rule with min-aggregation. The application of FGH-Rule is considered as an instance of query rewriting using views. This is an heuristics based approach and is currently implemented for linear programs.

Although the syntax of Datalog greatly differs from RA, the effects of magic-sets [Bancilhon et al., 1985, Saccà and Zaniolo, 1985, Gardarin, 1987] and of demand transformations [Tekle and Liu, 2011] are comparable to pushing certain kinds of selections and projections. These techniques are very sensitive to whether the Datalog program is written in a left-linear or right-linear manner, but one can use the automated reversal technique proposed in [Naughton et al., 1989] to fully exploit them.

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 [Jachiet et al., 2020] (detailed more in Section 2.6.3.3). 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 [Jachiet et al., 2020].

2.6.1.2 Automata based approaches.

Automata based approaches [Goldman and Widom, 1997] translate individual RPQs to automata and then join or union these results. This line of work considers in regular expressions. Regular expressions are possible to be expressed in the forms of an automaton. For a regular expression r it is possible to construct an automaton that recognizes the language of r. The query can be evaluated with an automaton and can manipulate tuples to show that there exists a path p from a_1 to a_2 and can be denoted as (a_1, a_2, p). A graph database can be converted into an automaton and show the possible paths between nodes. This is how a regular expression is expressed using an automaton.

2.6.2 Recursion in SQL

SQL is a standard query language that is used to manage and manipulate data in a RDBMS. Recursion in SQL is not represented by an algebraic operator. In fact, recursion is stands as "a barrier" in SQL. It means that it will be detected, but it cannot be optimized. There are no rewrite rules that would produce new plans considering the new information added to the query for each iteration of recursion.

Meanwhile, there are some other approaches that optimize recursion in relational algebra.

2.6.3 Extensions of relational algebra with recursion

α-RA [Agrawal, 1988] - is an extension of relational algebra which is noted as α. It is able to represent transitive closure $R+$ of a given term R. If we consider the expressive power over graphs, it corresponds to regular path queries (and their conjuctions and unions) but it struggles in expressing a^n/b^n queries (non-regular). It is not able to express a query that contains the same number n for a and b (in this case).

2.6.3.1 LFP-RA

LFP-RA [Aho and Ullman, 1979] - Another extension of relation algebra is LFP-RA. It is equipped with a "least fixpoint" construct. When considering the relation X and a given term R, the least fixpoint is calculated by adding the results in each iteration i when the relation X is filled by X_i. It has the same expressivity as linear Datalog (with stratified negation) as shown in [Jachiet et al., 2020].

2.6.3.2   Waveguide

Waveguide [Yakovets et al., 2015a] - A combination of finite state machines and α-RA, where plans are found by first starting on one side and matching the other side as well. The idea is to start from the left and then match the right one, or vice versa, first the right one and then the left one. The computation can even start in the middle and then go both ways to compute the RPQ. Other extensions to this work are made in [Abul-Basher et al., 2017a] where disjunctions of RPQs are treated, where some computations can be shared; and [Godfrey et al., 2017] that focuses on conjunctions of RPQs. But all these works are focused on conjuctions or unions of RPQs. It cannot express non-regular queries, for example: A^n/B^n.

2.6.3.3 μ-RA

μ-RA is an extension of relational algebra with a fixpoint operator to represent recursion. Below, we recall important definitions from [Jachiet et al., 2020].

2.6.3.4   Syntax of μ-RA

Below is the syntax of classical Relational Algebra introduced by Codd [Codd, 1970] equipped with the fixpoint operator introduced by [Jachiet et al., 2020].

a, b $\rightarrow$ columns

$$
\begin{array}{lll}
\varphi & ::= & \text{term} \\
 & |c \rightarrow v| & \text{constant} \\
 | & \sigma_\theta(\varphi) & \text{filter} \\
 | & \varphi_1 \bowtie \varphi_2 & \text{join} \\
 | & \varphi_1 \triangleright \varphi_2 & \text{antijoin} \\
 | & \varphi_1 \cup \varphi_2 & \text{union} \\
 | & \rho_a^b(\varphi) & \text{rename} \\
 | & \widetilde{\pi}_a(\varphi) & \text{antiprojection} \\
 | & X & \text{relation variable} \\
 | & \mu X.\, \varphi_{const} \cup \varphi_{rec} & \text{fixpoint}
\end{array}
$$

2.6.3.5   Preliminaries of [Jachiet et al., 2020]

Relations are considered as set of mappings that associate column names to values as mentioned in Section 2.2.2 of Relational Algebra.

2.6.3.6   Semantics

In this section, we show the semantics for each algebraic operator, including the fixpoint one.

Constant	$[\![|c \to v |]\!]_V$	$=$	$\{\{c \to v\}\}$
Relation Variable	$[\![X]\!]_V$	$=$	$V(X)$
Filter	$[\![\sigma_f(\varphi)]\!]_V$	$=$	$\{m \mid m \in [\![\varphi]\!]_V \wedge f(m) = \top\}$
Join	$[\![\varphi_1 \bowtie \varphi_2]\!]_V$	$=$	$\{m_1 + m_2 \mid m_1 \in [\![\varphi_1]\!]_V \wedge m_2 \in [\![\varphi_2]\!]_V \wedge m_1 \sim m_2\}$
AntiJoin	$[\![\varphi_1 \triangleright \varphi_2]\!]_V$	$=$	$\{m \in [\![\varphi_1]\!]_V \mid \forall m' \in [\![\varphi_2]\!]_V \ \neg(m' \sim m)\}$
Union	$[\![\varphi_1 \cup \varphi_2]\!]_V$	$=$	$[\![\varphi_1]\!]_V \cup [\![\varphi_2]\!]_V$
Rename	$[\![\rho_a^b(\varphi)]\!]_V$	$=$	$\{\{c \to v \in m \mid c \neq a\} \cup \{b \to v \mid a \to v \in m\} \mid m \in [\![\varphi]\!]_V\}$
AntiProjection	$[\![\widetilde{\pi}_a(\varphi)]\!]_V$	$=$	$\{\{c \to v \in m \mid c \neq a\} \mid m \in [\![\varphi]\!]_V\}$
Fixpoint	$[\![\mu(X = \varphi)]\!]_V$	$=$	$[\![X]\!]_{V[X/U_\infty]}$ where
		$=$	$U_0 = \emptyset, U_{i+1} = U_i \cup [\![\varphi]\!]_{V[X/U_i]}$, and $U_\infty = \bigcup_{n \in \mathrm{N}} U_i$

2.6.3.7 Type system

$FC(\theta)$ is the set of filtered columns based on filter f. Its definition is as follows:

- $FC(c = v) = \{c\}$

- $FC(\theta_1 \wedge \theta_2) = FC(\theta_1) \cup FC(\theta_2)$

- $FC(\theta_1 \vee \theta_2) = FC(\theta_1) \cup FC(\theta_2)$

- $FC(\neg\theta) = FC(\theta)$

- $FC(\theta_1 \ eq \ f_2) = FC(\theta_1) \cup FC(\theta_2)$

$$\Gamma \vdash |\mathsf{c} \mapsto \mathsf{v}| : \{\mathsf{c}\} \qquad \frac{\Gamma(X) = \tau}{\Gamma \vdash X : \tau} \qquad \frac{\Gamma \vdash \varphi_1 : \tau \qquad \Gamma \vdash \varphi_2 : \tau}{\Gamma \vdash \varphi_1 \cup \varphi_2 : \tau}$$

$$\frac{\Gamma \vdash \varphi_1 : \tau \qquad \Gamma \vdash \varphi_2 : \tau}{\Gamma \vdash \varphi_1 - \varphi_2 : \tau} \qquad \frac{\Gamma \vdash \varphi_1 : \tau_1 \qquad \Gamma \vdash \varphi_2 : \tau_2}{\Gamma \vdash \varphi_1 \bowtie \varphi_2 : \tau_1 \cup \tau_2}$$

$$\frac{\Gamma \vdash \varphi_1 : \tau_1 \qquad \Gamma \vdash \varphi_2 : \_}{\Gamma \vdash \varphi_1 \rhd \varphi_2 : \tau_1} \qquad \frac{\Gamma \vdash \varphi : \tau \qquad FC(\mathfrak{f}) \subseteq \tau}{\Gamma \vdash \sigma_{\mathfrak{f}}(\varphi) : \tau}$$

$$\frac{\Gamma \vdash \varphi : \tau \qquad a \in \tau \qquad b \notin \tau}{\Gamma \vdash \rho_{\mathsf{a}}^{\mathsf{b}}(\varphi) : (\tau \setminus \{\mathsf{a}\}) \cup \{\mathsf{b}\}} \qquad \frac{\Gamma \vdash \varphi : \tau \qquad a \in \tau}{\Gamma \vdash \widetilde{\pi}_a(\varphi) : t \setminus \{a\}}$$

$$\frac{\Gamma \vdash \kappa : \tau \qquad \Gamma \cup \{X \to \tau\} \vdash \psi : \tau}{\Gamma \vdash \mu X = (\kappa \cup \psi) : \tau}$$

Figure 2.7: Type rules

2.6.3.8 μ-RA properties

Definition 1. *Given a term φ, we say that φ is constant in X when X is not a free variable of φ.*

Definitions are taken from [Jachiet et al., 2020].

Below, we show the syntactical properties a fixpoint term being constant or recursive.

A fixpoint term $\mu(X = \varphi)$ is considered to be:

- *positive*: for all subterms $\varphi_1 \rhd \varphi_2$ of φ, φ_2 is constant in X;

- *linear*: for all subterms of φ of the form $\varphi_1 \bowtie \varphi_2$ or $\varphi_1 \rhd \varphi_2$, either φ_1 or φ_2 is constant in X;

- *mutually recursive*: there exists a subterm $\mu(X = \psi)$ of φ with X free in ψ.

Now we show the semantic interpretation of the above syntactical properties. When considering a term φ:

- If φ is recursive in X then for all V, $[\![\varphi]\!]_{V[X/\emptyset]} = \emptyset$.

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

A decomposed fixpoint term $\mu(X = \kappa \cup \psi)$ has its constant part κ constant in X and its recursive part ψ recursive in X.

Example   Considering a binary relation R represented as a table with two columns: {src, trg}. Its closure is the following:

$$\mu X. \ \ R \cup \widetilde{\pi}_m(\rho_{\text{trg}}^{\text{m}}(R) \bowtie \rho_{\text{src}}^{\text{m}}(X))$$

The above term is a decomposed fixpoint with R as its constant part and $\widetilde{\pi}_m(\rho_{\text{trg}}^{\text{m}}(R) \bowtie \rho_{\text{src}}^{\text{m}}(X))$ as its recursive part.
The type system for μ-RA concerning fixpoints shows that the type of the constant part of a fixpoint term is actually the type of its recursive part as well.

2.6.3.9   Auxiliary functions for rule application in μ-RA

In μ-RA, new rewrite rules concerning fixpoints are introduced. For these rewrite rules to be applied, some criteria need to be verified. These criteria are introduced in [Jachiet et al., 2020].
One criteria is based on the columns that are part or not of the iteration. A criteria called stabilizer is defined, which returns the set of columns that are not changed by the iteration. In order to find this set of columns, a set of derivations is defined. These derivations are shown in the follwing definition.

Definition 2. *The set of derivations $d(\psi, X)$ is:*

$$d(\psi_1 \cup \psi_2, X) \quad = \quad d(\psi_1, X) \cup d(\psi_2, X)$$

$$d(\psi_1 \triangleright \psi_2, X) \quad = \quad d(\psi_1, X)$$

$$d(\psi_1 \bowtie \psi_2, X) \quad = \quad d(\psi_1, X) \cup d(\psi_2, X)$$

$$d(\rho_a^b(\psi), X) \quad = \quad \{p \circ (b \to a, a \to \bot) \mid p \in d(\psi, X)\}$$

$$d(\widetilde{\pi}_a(\psi), X) \quad = \quad \{p \circ (a \to \bot) \mid p \in d(\psi, X)\}$$

$$d(\sigma_f(\psi), X) \quad = \quad d(\psi, X)$$

$$d(\mu(Y = \psi), X) \quad = \quad \emptyset$$

$$d(X, X) \quad = \quad \{()\} \ \textit{(a singleton identity)}$$

$$d(X, R) \quad = \quad \emptyset$$

$$d(|c \to c|, X) \quad = \quad \emptyset$$

Where $\circ$ repre-

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

Definition of stabilizer is as follows:

Definition 3. *Given a term φ linear and positive in a variable X, we define the stabilizer of X in ψ as the following set of column names: $stab(\psi, X) = \{c \in \mathfrak{C} \mid \forall p \in d(\psi, X)\ p(c) = c\}$.*

The other criteria defined for fixpoints is to check if a column can be added or removed when an iteration happens. These columns should play no role in the recursive computation of the term. This is calculated by the set of derivations that can be found in the following definition.

Definition 4. *We say that a column $c \in \mathfrak{C}$ can be added to or removed from a term $\psi \in \mathcal{F}[\Gamma]$ recursive in X when $add(\psi, X, c) = \top$ holds, with add defined as:*

$$add(\psi_1 \cup \psi_2, X, c) \quad = \quad add(\psi_1, X, c) \wedge add(\psi_2, X, c)$$

$$add(\psi_1 \bowtie \psi_2, X, c) \quad = \quad add(\psi_1, X, c) \wedge add(\psi_2, X, c)$$

$$add(\psi_1 \rhd \psi_2, X, c) \quad = \quad add(\psi_1, X, c) \wedge add(\psi_2, X, c)$$

$$add(\rho_a^b(\psi), X, c) \quad = \quad add(\psi, X, c) \wedge c \notin \{a, b\}$$

$$add(\widetilde{\pi}_a(\psi), X, c) \quad = \quad add(\psi, X, c) \text{ when } c \neq a$$

$$add(\widetilde{\pi}_c(\psi), X, c) \quad = \quad X \notin free(\psi)$$

$$add(\sigma_f(\psi), X, c) \quad = \quad add(\psi, X, c) \wedge c \notin FC(f)$$

$$add(\mu(Y = \psi), X, c) \quad = \quad add(\psi, X, c)$$

$$add(R, X, c) \quad = \quad c \notin \Gamma(R) \text{ when } X \neq R$$

$$add(X, X, c) \quad = \quad \top$$

$$add(|c' \rightarrow v|, X, c) \quad = \quad c \neq c'$$

2.6.4   Rewrite rules concerning fixpoint

Considering the syntax presented and the addition of the fixpoint operator in relational algebra, in [Jachiet et al., 2020] they propose new rewrite rules concerning fixpoints. These rewrite rules can be applied only when criteria using the auxiliary functions presented earlier are satisfied. We recall the rewrite rules containing criteria to be verified from [Jachiet et al., 2020]:

Theorem 1 (Pushing Filter inside a Fixpoint)**.** *Let $\mu(X = \kappa \cup \psi)$ be a decomposed fixpoint term, V an environment and f a filter condition with*

$FC(f) \subseteq stab(\psi, X)$. *Then we have* $[\![\sigma_f(\mu(X = \kappa \cup \psi))]\!]_V = [\![\mu(X = \sigma_f(\kappa) \cup \psi)]\!]_V$.

Theorem 2 (Pushing Anti-Join inside a Fixpoint). *Let* $\mu(X = \kappa \cup \psi)$ *be a decomposed fixpoint term, V an environment and ξ a term of type $t \subseteq stab(\psi, X)$ (we suppose that X is not a free variable of ξ). Then we have* $[\![\mu(X = \kappa \cup \psi) \cap \xi]\!]_V = [\![\mu(X = (\kappa \cap \xi) \cup \psi)]\!]_V$.

Theorem 3 (Pushing Join inside a Fixpoint). *Let* $\mu(X = \kappa \cup \psi)$ *be a decomposed fixpoint of type t_k, and $\varphi \in \mathcal{F}[\Gamma]$ (with $X \notin free(\varphi)$) a term of type t_φ such that:*
(1) $t_\varphi \subseteq stab(\psi, X)$
(2) $\forall c \in t_\varphi \setminus t_k \in add(\psi, X, c)$
Then we have $\Gamma \vdash \mu(X = \kappa \bowtie \varphi \cup \psi) : t_\varphi \cup t_\kappa$ *for all V compatible with Γ:*
$[\![\varphi \bowtie \mu(X = \kappa \cup \psi)]\!]_V = [\![\mu(X = \kappa \bowtie \varphi \cup \psi)]\!]_V$

Theorem 4 (Merging fixpoints). *Given two decomposed fixpoints $\mu(X = \kappa_1 \cup \psi_1)$ and $\mu(X = \kappa_2 \cup \psi_2)$ of types t_1 and t_2 such that:*
(1) $t_1 \cap t_2 \subseteq stab(\psi_2, X, C_2) \cap stab(\psi_1, X, C_1)$
(2) $\forall c \in t_1 \setminus t_2 \quad add(\psi_2, X, c)$
(3) $\forall c \in t_2 \setminus t_1 \quad add(\psi_1, X, c)$ *then we have:*
$[\![\mu(X = \kappa_1 \cup \psi_1) \bowtie \mu(X = \kappa_2 \cup \psi_2)]\!]_V = [\![\mu(X = \kappa_1 \bowtie \kappa_2 \cup \psi_1 \cup \psi_2)]\!]_V$.

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

2.6.5 Application of μ-RA for recursive query optimization on knowledge graphs

One possible application of μ-RA is the optimization of queries on knowledge graphs. We start by explaining how knowledge graphs can be represented in RA's data model. A knowledge graph can be represented by a table that contains all the graph edges. This table has three columns: source and target nodes and one column, label, for the predicate that labels the edge.

Figure 2.8 illustrates an example of a relation of this type.

src	label	trg
Jean	livesIn	Paris
France	isLocatedIn	Europe
Jim	worksAt	Inria
Jim	knows	Jean
...	...	...

Figure 2.8: Representation of a knowledge graph in μ-RA.

To extract information from these graphs we can write queries in the form of a UCRPQ. We show the following example:

$$?y \leftarrow \ ?x : \texttt{knows+} \ \ ?y : \texttt{Jean}$$

This query returns the totality of people that know Jean directly or indirectly.

2.7   Summary

Above we introduced the main approaches (algebraic or not) that consider recursion. Among those there are several that treat the transitive closure and the main optimization rules like: the possibility to push selection and projection in the fixpoint.

In terms of expressivity there is one approach that stands among them all. This is μ-RA, the approach presented in [Jachiet et al., 2020]. Thanks to the rewrite rules made possible from this addition, in particular the possibility of merging fixpoints which can be expressed as the path $(a + /b+)$, it is possible to reach new plans that were not reachable before. This is why the plan space is bigger. Since for the other approaches it is not possible to reach these plans, we focus on μ-RA approach.

Limitations of μ-RA. μ-RA [Jachiet et al., 2020] provides a rich set of rewrite rules for recursive terms enabling efficient recursive evaluation plans not available with earlier approaches. However [Jachiet et al., 2020] does not study plan enumeration with the new rules.

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

With recursion, plan spaces are often significantly larger than in the non-recursive setting, due to new interplays between recursive and non-recursive operators. The efficiency of recursive plan enumeration becomes critical. The speed of plan enumeration directly determines whether query evaluation plans enabled by, e.g. μ-RA [Jachiet et al., 2020], are within range or theoretically reachable, but still out of reach for a practical query optimizer. This motivates the search for efficient methods for enumerating recursive plans.

Limitations of plan enumeration approaches and of the LQDAG approach in particular.   The different approaches found in the litterature, either bottom-up or top-down, are mostly interested in join enumeration optimization. No previous approach treats recursion.

LQDAG is a factorized representation used for recursion-free queries. No earlier work treats recursion in a LQDAG based approach.

Specific problem considered in this thesis. This thesis seeks methods for the efficient enumeration of recursive plans.

2.8 Challenges in extending the LQDAG to support Recursion

In Section 2.3.6 we explained how the optimization of SPJ queries are handled in query optimization. In the classical SPJ setting, selections and projections do not add to the combinatorics and this is why the litterature focuses on join enumeration. We will refer to some definitions of Section 2.3.6 to explain the different stages of optimization in SPJ setting. For an AST with only (select-project-join) SPJ the optimization is done following the example illustrated in Figure 2.4 for each stage. We discussed about the strict type system and showed an example there.
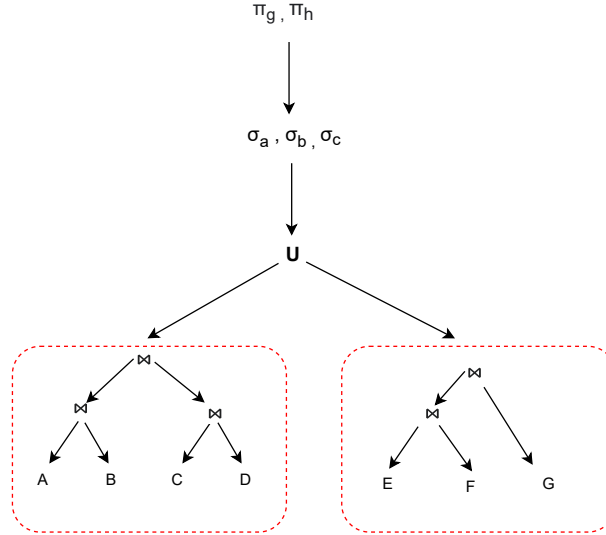
Figure 2.9: Union as a binary operator in the optimization phase.

Now we will continue the discussion where we left off in Section 2.3.6. Starting from an initial query where the operators such as: join, selection, projection and union can be found anywhere, can we achieve an AST as in Figure 2.9 without breaking the strict typing rule:

$$\frac{\Gamma \vdash \varphi_1 : \tau \qquad \Gamma \vdash \varphi_2 : \tau}{\Gamma \vdash \varphi_1 \cup \varphi_2 : \tau}$$

It is not immediate to transfer all the results considering join enumeration to queries that include union operator as well. Selections and projections cannot be easily pushed without breaking the types of the query if we use the same strict type system. The example of Figure 2.9 is a proposition of the napplication of optimization methods used in the litterature for join enumeration separately to

lower the complexity to the maximum. To consider this we have to unplug the type system and allow the selections projections and unions to be pulled, no matter where they can be found in the AST. Considering the strict type system, for union to be considered correct, only operands of the same type are allowed. This is a barrier for the pulling stage to happen. This means that if we need to allow this phase of pulling we need to modify the type system. What will be the consequences of this adaptation? Below, we propose an adapted type system for union:

$$\frac{\Gamma \vdash \varphi_1 : \tau' \qquad \Gamma \vdash \varphi_2 : \tau''}{\Gamma \vdash \varphi_1 \cup \varphi_2 : \tau' \cup \tau''}$$

This typing rule adaptation allows us to consider union as correct even when *type* of φ_1 is different to the *type* of φ_2. After applying apply the pulling stage of selections, projections and unions we are in a situation like the one presented in Figure 2.9. The rewrite rules allowed here are (without restrictions): Pulling Selection from Join is as follows: $\varphi \bowtie \sigma_\theta(\psi) \to \sigma_\theta(\varphi \bowtie \psi)$; Pulling Projection from Join: $\sigma_f(\varphi) \bowtie \psi \to \sigma_f(\varphi \bowtie \psi)$; Pulling Union from Join: $(\varphi \bowtie \psi) \cup (\varphi \bowtie \gamma) \to \varphi \bowtie (\psi \cup \gamma)$; Pulling Selection from Union is as follows: $\varphi \cup \sigma_\theta(\psi) \to \sigma_\theta(\varphi \cup \psi)$; Pulling Projection from Union: $\sigma_f(\varphi) \cup \psi \to \sigma_f(\varphi \cup \psi)$. One benefit of the adapted type system is that we could be able to optimize the join enumeration, for example using an algorithm as in [Shanbhag and Sudarshan, 2014] used for top-down approaches and then push all the other operators to the maximum.

Challenges for adding recursion   Let's consider adding the fixpoint operator that represents recursion. Like in the case of union, we will consider a less strict type system:

$$\frac{\Gamma \vdash \kappa : \tau' \qquad \Gamma \cup \{X \to \tau\} \vdash \psi : \tau''}{\Gamma \vdash \mu X = (\kappa \cup \psi) : \tau' \cup \tau''}$$

Selections and projections would be allowed to be pulled at a first stage. The considered rewrite rules would be the following (without restrictions):
Pulling Selection from Fixpoint: $\mu(X.\sigma_f(\kappa) \cup \psi) \to \sigma_f(\mu(X = \kappa \cup \psi))$
Pulling Projection from Fixpoint: $\mu(X.\pi_c((\kappa \cup \psi))) \to \pi_c((\mu(X = \kappa \cup \psi)))$.
Given that in the case of recursion new columns are presented for each iteration, allowing this stage of pulling is not "safe". For each iteration of the recursion we would not be certain for the columns present or that are possible to be introduced at a given time. We would not keep count on the columns being added and cannot control the type of the fixpoint. To ensure the validity of rewrite rule application there has to be some stricter type system that allows checking the columns that we are certainly present and columns that might be added to the recursion. Anyways, the above adaptation of type system when introducing fixpoints needs to be checked in practice as well.

A stricter type system, which is used in recursion in this study is the following:

$$\frac{\Gamma \vdash \varphi_1 : \tau \qquad \Gamma \vdash \varphi_2 : \tau}{\Gamma \vdash \varphi_1 \cup \varphi_2 : \tau}$$

In conclusion, each approach has its own difficulties. On one hand, without a strict type system we would lose control over the columns that are present, or will be introduced at a given time, but allow a proper stage of operator pulling. On the other hand, with the strong type system we cannot allow a stage of operator pulling, but we have more control over columns that are present for each iteration.

In the next chapter, we present the main contribution of this thesis which proposes an approach for efficiently enumerating recursive plan spaces, keeping the strict type rules for union and fixpoint.

Part II

Contribution

Summary and Outline of Contribution

In the previous state-of-the-art part, we outlined the interests of the rich plan spaces recently made possible by the μ-RA approach for recursive queries. We also outlined the current limits and difficulties for efficiently enumerating non-recursive plan spaces in actual query optimizers.

A clear gap remains: how to benefit from rich plan spaces for recursive queries in actual query optimizers? Or in other terms, how to extend the most advanced plan enumeration methods so as to efficiently enumerate recursive plans?

This part of the manuscript describes the contribution of this thesis, which aims at filling this gap. The contribution part is structured into two chapters:

Chapter 3 presents the theory and algorithmic techniques of a **new RLQDAG approach**. Contributions include the first extension of the LQDAG with the support of recursive queries; the first formalization of important LQDAG concepts in terms of formal syntax and formal semantics, with a particular focus on the sharing of common subterms and recursion; generalized rewrite rules enabling grouped transformations of sets of recursive terms (instead of individual terms); and the complete technical machinery with algorithmic techniques for their implementation, including the incremental propagation of annotations required for guiding transformations of sets of recursive subterms.

Chapter 4 reports on an experimental assessment of this new approach applied in the setting of recursive graph queries. Contributions include a complete prototype implementation of the RLQDAG; an application for property graph queries, including a direct translation of a recursive query language fragment for property graphs into RLQDAG; and lessons learned from extensive practical experiments using synthetic and real datasets.

Chapter 3

RLQDAG

3.1 Introduction

In this Chapter we present the main contribution of this thesis which is a novel method for efficiently enumerating plan spaces in recursive relational algebra.

Specifically, we propose an extended logical query DAG, named *recursive logical query dag (RLQDAG)*, to support recursive algebraic terms. The main purpose of the RLQDAG is to introduce a compact representation of recursive terms that allows for their rewriting as sets of terms instead of individual terms. The goal is to enable a much faster exploration of equivalent recursive plans. This exploration phase is crucial as it may produce recursive terms which are drastically more efficient. The faster we generate the space of equivalent plans, the more likely we will be able to find terms with more efficient evaluation. Ultimately, if we manage to explore the entire space of recursive plans, we find the optimal term faster. This is particularly important for recursive terms since their rule sets usually generate huge plan spaces. It is common practice to allocate a time budget for this exploration phase, as it is notoriously known that exhaustive explorations may not be feasible in practice for certain queries. The exploration phase is a prerequisite for query evaluation in any transformation-based query optimizer that first needs to find an optimal – or best estimated – term. For all these reasons, the speed of plan space exploration represents one of the most critical aspect in recursive query optimization.

Outline. We first give intuitions of the RLQDAG before we introduce it formally. We present a syntax and semantics for the RLQDAG. To the best of our knowledge, it is the first time a syntax and formal semantics are defined for LQDAG. This is useful to facilitate the development and presentation of the theory of sharing subterms in the presence of recursion. In particular it is useful for presenting the generalized rewrite rules that enable the transformation of sets of recursive subterms at once (instead of individual subterms); to facilitate proofs, and to describe the transformations of RLQDAG subparts more precisely and more rigorously.

3.2   Recursive structure in the RLQDAG: principles

The RLQDAG extends the LQDAG with the ability to capture and transform sets of recursive terms. We introduce a binary operator that models the μ decomposed fixpoint operator, as illustrated in Figure 3.1.
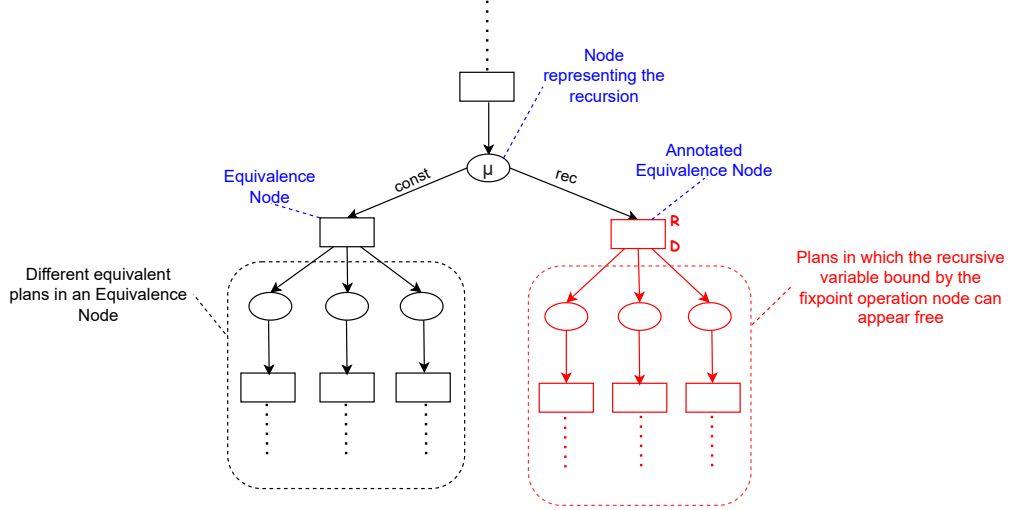
Figure 3.1: Structure of a recursive term in the RLQDAG.

One of the operands models the constant part of the fixpoint operator. It is expressed as an equivalence node that regroups all the semantically equivalent subterms of the constant part. The other operand models the recursive part. The major difference brought by the recursive part is that in this branch we will find at least one explicit occurrence of the recursive variable which is bound by the fixpoint operator. This aspect will lead to a number of discussions, choices, new definitions and developments. This is because depending on how the recursive variable is used in that branch,the transformation and sharing of subterms may – or may not – be allowed. All these concepts will be described in the following sections.

Using the representation illustrated in 3.1, we will define and apply rewrite rules that will transform and create new fixpoint operators, with their respective constant and recursive parts, while maximizing the sharing of the existing subparts.

For example, Figure 3.2 and Figure 3.3 illustrate sample RLQDAG structures obtained after the process of *expansion*, that consists in populating the space of equivalent plans obtained by applying possible transformations. Expanded RLQDAGs can represent large sets of algebraic terms, while sharing common subterms. Figures 3.2 and 3.3 respectively illustrate sample expanded RLQDAGs obtained for two UCRPQ queries taken from [Jachiet et al., 2020]. The query corresponding to Figure 3.2 is:

```
?area  ←  <wikicategory_Capitals_in_Europe>
          -rdf:type/(<isLocatedIn>+/<dealsWith>|<dealsWith>)
          ?area
```

and the query corresponding to Figure 3.3 is:

```
?a ←    ?a <isLocatedIn>+/<isConnectedTo>+/<dealsWith>+ <Japan>
```

The RLQDAG shown in Figure 3.2 represents 296 terms, and the one of Figure 3.2 represents 436 terms. Some equivalence nodes are shared between several terms. This can be noticed by the many arrows that point to the same equivalence nodes. The rewrite rules used to generate these factorized representations of plan spaces are the ones of relational algebra described in Section 2.2.4 adapted to operate on the RLQDAG, and the ones concerning fixpoints that will be presented in Section 3.6.

In the sequel, we introduce the RLQDAG more formally.

Figure 3.2: Expanded RLQDAG example.
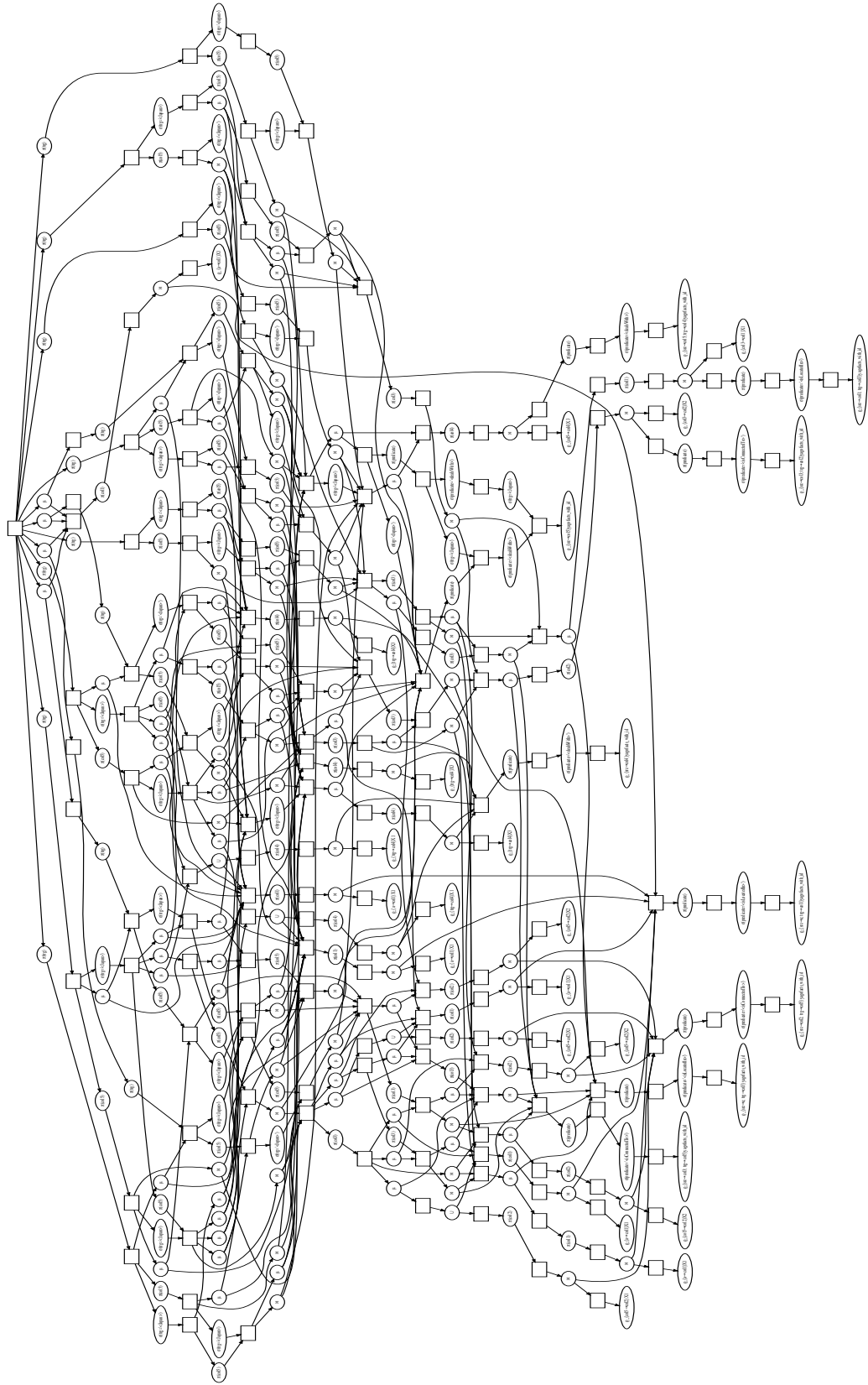
Figure 3.3: Another expanded RLQDAG example.

3.3   Syntax of RLQDAG terms

The syntax of RLQDAG terms is presented in Figure 3.4, where a, b denote column names and f is a filter expression. The entrypoint syntactic construct is an equivalence node $[\alpha]$. This means that every RLQDAG is an $[\alpha]$ at top-level.

An equivalence node is a node that regroups several operation nodes d, 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 name Y to α_1, and allows Y to be used multiple times in α_2 as a reference to α_1. The general definition of an equivalence node 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. Operation nodes include the main algebraic operations of recursive relational algebra. Each operand of an operation node d is in turn an equivalence node γ.

The rename operator $\rho_{\mathsf{a}}^{\mathsf{b}}(\gamma)$ renames column a into column b in the equivalence node[1]. 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.

Recursive terms can be expressed using fixpoint operation nodes. The principle, inspired from earlier works in recursive relational algebras [Aho and Ullman, 1979, Jachiet et al., 2020], consists in the introduction of a least-fixpoint binder operation node (μ) that binds a fresh variable X to some expression in which the variable X can appear, thus explicitly denoting recursion. In the RLQDAG, as defined in the abstract syntax of Fig. 3.4 and illustrated in Fig. 3.1, a fixpoint operator node is written $\mu X.\ \gamma \cup \alpha_{rec}$. The first operand γ is an equivalence node that models the constant part (the base case) of the recursion. X cannot occur within γ. The other equivalence node α_{rec} is the recursive part. An important aspect is that α_{rec} contains at least one free occurrence of the recursive variable X. This is a major difference between the fixpoint operation node and the other operation nodes. This aspect will lead to a number of new definitions and formal developments. This is because depending on how the recursive variable is used in that branch, the transformation and sharing of subterms in the RLQDAG may, or may not, be allowed.

For example, the following RLQDAG term corresponds to the transitive closure of some relation A:

$$\mu X.\ [A] \cup \big[\ [A] \bowtie [X]\ \big]_{\mathfrak{D}}^{\mathfrak{R}}$$

In this case, the equivalence node for the constant part contains the relation variable A, and the equivalence node for the recursive part is composed of the join between A and X. In a RLQDAG, the recursive equivalence node of each fixpoint operation node is annotated (with annotations $\mathfrak{D}$ and $\mathfrak{R}$). These annotations will

---

[1]As a slight discrepancy between the theory and its implementation, in the prototype implementation we automatically push, combine and simplify renamings so that they appear only in front of variables X. So the core syntax for the rename operator in the implementation is $\rho^{\{a \mapsto b, \dots, w \mapsto z\}}(X)$. The implementation also provides a way to denote $\rho_{\mathsf{a}}^{\mathsf{b}}(\gamma)$ as a syntactic sugar that is automatically translated into the core syntax.

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
	|	$\gamma \rhd \gamma'$	Antijoin
	|	$\gamma \cup \gamma'$	Union
	|	$\widetilde{\pi}_a(\gamma)$	Antiprojection
	|	$\mu X. \gamma \cup \alpha_{rec}$	Fixpoint (recursion)

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

Figure 3.4: Syntax of RLQDAG terms.

be useful for guiding the application of transformations and will be defined and explained in Section 3.5.1.

One major difference between the fixpoint operator and the other binary operators is that terms present in the recursive part of the fixpoint (α_{rec}) contain free occurences of the recursive variable. There are two reasons why we distinguish α_{rec} from a general equivalence node α in the abstract syntax. The first reason is that those equivalence nodes for recursive parts are equipped with annotations (detailed in Section 3.5.2.) that will be useful to control subterm transformation and in particular to guide the application of generalized rewrite rules. 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.

Furthermore, we consider the following (usual) restrictions over the abstract syntax presented in Figure 3.4: we consider only positive, linear and non mutually recursive RLQDAG terms:

- positive means that recursive variables only appear in the left-hand operand of an antijoin;
- linear means that one of the operands in a join or antijoin operation is constant in the free variable;
- Non-mutually recursive terms means that fixpoint terms are properly nested in such a way that there is only one free variable in any fixpoint subterm (this does not prevent this same variable to occur one or more times).

3.4   Semantics of RLQDAG terms

The interpretation of a RLQDAG term is the set of all μ-RA terms that it represents. Formally, the semantics of a RLQDAG $[\alpha]$ is given by the functions $S_\alpha[\![]\!]$ and $S_\gamma[\![]\!]$ presented in Figure 3.5, where E denotes a variable environment used to keep track of the variable definitions introduced by binders for the sharing of subterms. $S_\alpha[\![\alpha]\!]_\emptyset$ returns the interpretation of a RLQDAG $[\alpha]$.

Notion of well-formedness.   A well-formed RLQDAG is a RLQDAG whose interpretation is a set of semantically equivalent μ-RA terms:

Definition 5 (Well-formedness)**.** *A RLQDAG $\big[\alpha\big]$ is well-formed if and only if* $\forall t, t' \in S_\alpha[\![\alpha]\!]_\emptyset, \quad [\![t]\!]_\emptyset = [\![t']\!]_\emptyset.$

In this definition $[\![t]\!]_\emptyset$ returns the interpretation of the individual recursive relational algebraic term t (i.e. the set of mappings returned by t; or in other terms the resulting relational table of t), as returned by the formal semantics function $[\![t]\!]_V$ of μ-RA terms defined in Chapter 2.

Types.   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*:

Relation Variable	$S_d[\![X]\!]_E$	$=$	$\{X\}$
Filter	$S_d[\![\sigma_f(\gamma)]\!]_E$	$=$	$\{\,\sigma_f(t) \mid t \,\in\, S_\gamma[\![\gamma]\!]_E\}$
Join	$S_d[\![\gamma_1 \bowtie \gamma_2]\!]_E$	$=$	$\{\,t_1 \bowtie t_2 \mid t_1 \in\, S_\gamma[\![\gamma_1]\!]_E \wedge t_2 \in\, S_\gamma[\![\gamma_2]\!]_E\}$
AntiJoin	$S_d[\![\gamma_1 \rhd \gamma_2]\!]_E$	$=$	$\{\,t_1 \rhd t_2 \mid t_1 \in\, S_\gamma[\![\gamma_1]\!]_E \wedge t_2 \in\, S_\gamma[\![\gamma_2]\!]_E\}$
Union	$S_d[\![\gamma_1 \cup \gamma_2]\!]_E$	$=$	$\{\,t_1 \cup t_2 \mid t_1 \in\, S_\gamma[\![\gamma_1]\!]_E \wedge t_2 \in\, S_\gamma[\![\gamma_2]\!]_E\}$
Rename	$S_d[\![\rho_a^b(\gamma)]\!]_E$	$=$	$\{\,\rho_a^b(t) \mid t \,\in\, S_\gamma[\![\gamma]\!]_E\}$
AntiProjection	$S_d[\![\widetilde{\pi}_a(\gamma)]\!]_E$	$=$	$\{\,\widetilde{\pi}_a(t) \mid t \,\in\, S_\gamma[\![\gamma]\!]_E\}$
Fixpoint	$S_d[\![\mu X.\ \gamma\ \cup\ \alpha_{rec}]\!]_E$	$=$	$\{\mu X.\ t\ \cup\ t_{rec} \mid t \,\in\, S_\gamma[\![\gamma]\!]_E$
			$\wedge\ t_{rec}\ \in\ S_\alpha[\![\alpha_{rec}]\!]_E\}$
Equivalence node	$S_\alpha[\![d]\!]_E$	$=$	$S_d[\![d]\!]_E$
Equivalence node	$S_\alpha[\![d, \alpha]\!]_E$	$=$	$S_d[\![d]\!]_E \cup S_\alpha[\![\alpha]\!]_E$
Equivalence node	$S_\alpha[\![\text{let } Y = \alpha_1 \text{ in } \alpha_2]\!]_E$	$=$	$S_\alpha[\![\alpha_2]\!]_{E'}$ with $E' = E \oplus \{y \mapsto \alpha_1\}$
Equivalence node	$S_\gamma[\![[\alpha]]\!]_E$	$=$	$S_\alpha[\![\alpha]\!]_E$
Equivalence node	$S_\gamma[\![Y]\!]_E$	$=$	$S_\alpha[\![E(Y)]\!]_E$

Figure 3.5: Formal semantics of RLQDAG terms.

$type(\gamma)$ 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.

3.5   Recursive terms and rule applicability

A significant novelty introduced by recursion when compared to the classical setting of non-recursive algebraic terms resides in the criteria used to trigger rewrite rules. In the classical non-recursive setting these criteria are trivial in the sense that they only depend on top-level operators. For example, when applying join distributivity over union, the applicability of the rewrite rule $A \bowtie (B \cup C) \longrightarrow (A \bowtie B) \cup (A \bowtie C)$ can be determined by examining only the combination of the two top-most operators, i.e. the top-level ($\bowtie$) with the operator immediately underneath (i.e. $\cup$). 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 any case of the non-recursive setting, these verifications need only to look at the two top-most algebraic operators under scrutiny (and their potential immediate parameters). No further traversal of subtrees of operators are required. For instance, in the previous example of join distributivity over union, B and C do not need to be traversed at all when determining rule applicability.

In sharp contrast, rules for transforming recursive terms rely on criteria that are significantly more sophisticated as they sometimes require a whole traversal of the recursive part of a fixpoint term. This is because opportunities for rule application with recursive terms depend on how the recursive variable is used within the recursive parts of fixpoints. It is known since the works of [Aho and Ullman, 1979, Kifer and Lozinskii, 1990, Jachiet et al., 2020] that criteria for rule application are significantly more sophisticated in the presence of recursion as they need to examine how the recursive variables are used. We show that it is still possible to apply rules over sets of recursive terms at once, using the concept of *annotated equivalence nodes*, that we define in § 3.5.2, and for which we need some preliminary definitions.

3.5.1   Preliminary definitions of auxiliary functions in RLQDAG

Definition 6 (Unfolding). *Let α be an equivalence node. The unfolding of α, denoted $\mathtt{unfold}(\alpha)$ 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, that generalize the definitions for individual algebraic terms found in [Jachiet et al., 2020]. These functions will be used for defining annotated equivalence nodes in Section 3.5.2.

Inspired from [Jachiet et al., 2020] we define the destabilizer ($\mathtt{destab}()$) as the set of columns that can be changed during an iteration of the fixpoint. $\mathtt{destab}()$ can be seen as the complement of the stabilizer calculated for individual terms in μ-RA [Jachiet et al., 2020], which is generalized for RLQDAG terms (i.e. sets of μ-RA terms). The calculation of $\mathtt{destab}()$ relies on a set of derivations, and it needs to traverse term subtrees in order to analyze how the occurrence of free variables are used.

Definition 7. *For a fixpoint operation node $\mu X.\ \gamma\ \cup\ \alpha_{rec}$ we consider $\alpha' = \mathtt{unfold}(\alpha_{rec})$ and we define $\mathtt{destab}(\alpha', X)$ as the following set of column names:*

$$\mathtt{destab}(\alpha', X) = \{c \in \mathfrak{C} \mid \exists p \in \mathtt{d}(\alpha', X)\ \ p(c) \neq c\}$$

where $\mathtt{d}(\cdot, \cdot)$ computes the set of derivations [Jachiet et al., 2020] over a RLQDAG term:

$$\mathtt{d}((d, \alpha), X) \quad\quad\quad\quad = \quad \mathtt{d}(d, X)$$

$$\mathtt{d}([\alpha_1] \cup [\alpha_2], X) \quad\quad = \quad \mathtt{d}(\alpha_1, X) \cup \mathtt{d}(\alpha_2, X)$$

$$\mathtt{d}([\alpha_1] \triangleright [\alpha_2], X) \quad\quad = \quad \mathtt{d}(\alpha_1, X)$$

$$\mathtt{d}([\alpha_1] \bowtie [\alpha_2], X) \quad\quad = \quad \mathtt{d}(\alpha_1, X) \cup \mathtt{d}(\alpha_2, X)$$

$$\mathtt{d}(\rho_a^b([\alpha]), X) \quad\quad\quad = \quad \{p \circ (b \to a, a \to \perp) \mid p \in \mathtt{d}(\alpha, X)\}$$

$$\mathtt{d}(\widetilde{\pi}_a([\alpha]), X) \quad\quad\quad = \quad \{p \circ (a \to \perp) \mid p \in \mathtt{d}(\alpha, X)\}$$

$$\mathtt{d}(\sigma_f([\alpha]), X) \quad\quad\quad = \quad \mathtt{d}(\alpha, X)$$

$$\mathtt{d}(\mu(Z.\, \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}}), X) \quad = \quad \emptyset$$

$$\mathtt{d}(X, X) \quad\quad\quad\quad\quad = \quad \{()\} \;\; \text{(a singleton identity)}$$

$$\mathtt{d}(R, X) \quad\quad\quad\quad\quad = \quad \emptyset$$

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.

Notion of rigidity in RLQDAG We define a function $\mathtt{rigid}\,()$ that computes the set of columns that cannot be added nor removed from a fixpoint operation node. Intuitively, $\mathtt{rigid}\,()$ can be thought as the "complement" of the boolean predicate *add* (introduced for individual terms in μ-RA [Jachiet et al., 2020]) but which is generalized for a RLQDAG term, and returns a set of columns (instead of a boolean). A column $c \in \mathfrak{C}$ cannot be added nor removed from an annotated equivalence node α_{rec} (recursive in X) when $c \in \mathtt{rigid}(\mathtt{unfold}(\alpha), X)$ and $\mathtt{rigid}\,()$ is defined as follows:

Definition 8 (Rigidity).

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

3.5.2   Annotated equivalence node

An annotated equivalence node (α_{rec} in the abstract syntax of RLQDAG terms given in Fig. 3.4) 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 9. *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* $\mathfrak{R} = \texttt{rigid}(\alpha, X)$.

Annotations $\mathfrak{D}$ and $\mathfrak{R}$ are intended to characterize all the subterms of the annotated equivalence node (thanks to definition 10). The goal of annotated equivalence nodes is to guide and maximize the grouped application of transformations, while also maximizing the sharing of common subterms.

In the sequel, we detail RLQDAG transformations, by introducing new rewrite rules that are capable of transforming sets of subterms at once.

Intuition. Annotations play a crucial role for the application of rewrite rules in RLQDAG. Rewrite rules apply only for RLQDAG branches that satisfy criteria depending on annotations. For example, in Figure 3.6, an initial RLQDAG is shown with two initial terms (in black color). Annotations $\mathfrak{D}, \mathfrak{R}$ and $\mathfrak{D}', \mathfrak{R}'$ are shown on the Figure for two annotated equivalence nodes. It may happen that some of these annotations prevent rule applicability. For example here $\mathfrak{D}$ and $\mathfrak{R}$ do not allow the join operator to be pushed in the fixpoint of the first term, whereas for the second term, annotations $\mathfrak{D}'$ and $\mathfrak{R}'$ may allow it to be pushed. Annotations let us guide rewrite rule application in RLQDAG. By this rule application, the term shown in green color is generated and added to the set of terms of the same equivalence node as the first two terms. The new term is created, and some of its subparts are shared. Another annotated equivalence node is created with potentially different annotations $\mathfrak{D}''$ and $\mathfrak{R}''$. In the sequel, we detail RLQDAG transformations, by introducing new rewrite rules that are capable of transforming sets of subterms at once.
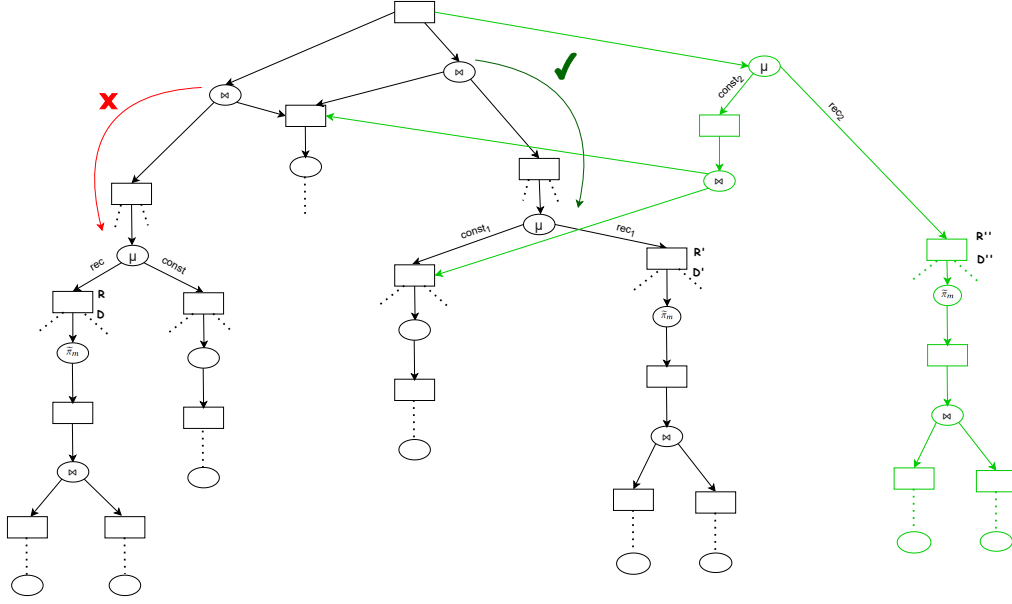
Figure 3.6: Sample of a recursive term and rewrite rule application.

Notion of consistency Intuitively, a RLQDAG is *consistent* iff it is well-formed and in addition, for any of its 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 10 (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 $\boldsymbol{cons}(\alpha_2, X)_{\mathfrak{D}}^{\mathfrak{R}}$*

$$\boldsymbol{cons}([d], X)_{\mathfrak{D}}^{\mathfrak{R}} = \boldsymbol{destab}(d, X) = \mathfrak{D}$$

$$\wedge \ \boldsymbol{rigid}(d, X) = \mathfrak{R}$$

where: $\quad \boldsymbol{cons}([d, \alpha], X)_{\mathfrak{D}}^{\mathfrak{R}} = \boldsymbol{destab}(d, X) = \mathfrak{D}$

$$\wedge \ \boldsymbol{rigid}(d, X) = \mathfrak{R}$$

$$\wedge \ \boldsymbol{cons}([\alpha], X)_{\mathfrak{D}}^{\mathfrak{R}}$$

In the remaining, we only consider consistent RLQDAGs[2].

---

[2]In particular in Chapter 4 where we introduce functions that generate RLQDAGs, we pay attention that all generated RLQDAGs are consistent RLQDAGs.

3.6   Generalized rewrite rules for transforming sets of recursive terms

We now propose RLQDAG transformations whose purpose is to efficiently build the space of equivalent recursive plans. RLQDAG transformations generalize the rewrite rules proposed in [Jachiet et al., 2020] for individual μ-RA terms so that they can operate on the RLQDAG representation: i.e. on sets of μ-RA terms at once (instead of successively on individual terms). RLQDAG transformations leverage annotated equivalence nodes to guide the transformation of 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 fixpoint transformation operation, we describe which parts are shared, how new 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 opportunties for transformations that are also explored.

For each fixpoint transformation operation, we show which parts are shared, where new terms created are added and what happens with the other plans already present in the equivalence node. Even though some subterms are not changed by the rewrite rule, some require to be explored in search for more opportunities for transformation. Furthermore, the creation of new combinations of operators may in turn generate more opportunties for transformations that also need to be properly explored.

We formalize all these ideas by introducing RLQDAG rewrite rules, based on the syntax of RLQDAG terms introduced in Section 3.3. 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. 3.7 focuses on the two main cases that correspond to potential opportunities of pushing filters, i.e. the cases $\mathrm{pf}(\lceil d \rfloor)$ and $\mathrm{pf}(\lceil d, \alpha \rfloor)$ 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[3].

---

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

$$\mathrm{pf}(\mu X.\ \gamma \cup \left\lceil \alpha \right\rfloor_{\mathfrak{D}}^{\mathfrak{R}}) \quad = \quad \mu X.\ \mathtt{expand}(\gamma) \cup \left\lceil \mathtt{expand}(\alpha) \right\rfloor_{\mathfrak{D}}^{\mathfrak{R}}$$

$$\mathrm{pf}(\lceil \left\lceil A \right\rfloor \bowtie \left\lceil B \right\rfloor \rfloor) \quad = \quad \mathtt{expand}(\left\lceil A \right\rfloor) \bowtie \mathtt{expand}(\left\lceil B \right\rfloor)$$

where the $\mathtt{expand}()$ function, defined in Section 3.7, simply traverses subterms in search for more transformation opportunities. pf() is defined similarly for all the other syntactic cases of γ.

$$\mathrm{pf}(\big[\sigma_f([\mu X.\ \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}}])\big]) =$$

$$\begin{cases} \bullet \quad \big[\mu X'.\ \mathrm{expand}([\sigma_f(\gamma)]) \cup \big[\mathrm{expand}(\alpha_{\{X/X'\}})\big]_{\mathfrak{D}}^{\mathfrak{R}'}\ \big] \\[2ex] \qquad when\ \ \mathtt{filt}(f) \subseteq \mathfrak{D} \\[2ex] \bullet \quad \big[\ \sigma_f(\mathrm{expand}(\big[\mu X.\ \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}}\big]))\ \big]\ otherwise \end{cases}$$

$$\mathrm{pf}(\big[\sigma_f([\mu X.\ \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}},\ \alpha_2])\big]) =$$

$$\begin{cases} \bullet \quad \big[\mu X'.\ \mathrm{expand}([\sigma_f(\gamma)]) \cup \big[\mathrm{expand}(\alpha_{\{X/X'\}})\big]_{\mathfrak{D}}^{\mathfrak{R}'}, \\[2ex] \qquad \mathrm{expand}(\big[\sigma_f([\alpha_2])\big])\ \big] \\[2ex] \qquad when\ \ filt(f) \subseteq \mathfrak{D} \\[2ex] \bullet \quad \big[\ \sigma_f(\mathrm{expand}(\big[\mu X.\ \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}},\ \alpha_2\big])),\ \\[2ex] \qquad \mathrm{expand}(\big[\sigma_f([\alpha_2])\big])\ \big]\ otherwise \end{cases}$$

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

Figure 3.7: Pushing a filter in an equivalence node containing a fixpoint operation node.

In Fig. 3.7, whenever the filter can be pushed through the fixpoint operation node, pf() generates a new RLQDAG in which the filter operation node is put within the constant part of the fixpoint operation node. In the resulting equivalence node, the initial (suboptimal) term is replaced by the new term where filtering is done earlier (hence more efficient). For this transformation to happen, the criteria (on the last line of Fig. 3.7) must be satisfied. Whenever it is not the case, the RLQDAG structure is not reordered but simply recursively traversed in search for more transformation opportunites. The creation of new combinations of operation nodes may in turn provide new opportunities for other rewritings (for instance, new opportunities for pushing filters even further, or even other kinds of rewritings). This is the role of the expand() function, formally defined in section 3.7. 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.

Fig. 3.8 illustrates graphically a RLQDAG before transformation, and Fig. 3.9 depicts its updated structure obtained after the transformation defined in Fig. 3.7.

Notice that when γ is a reference Y, there is no need to introduce a new binder, the reference name is used directly.

Figure 3.8: Initial RLQDAG before expansion with opportunity to apply pf().
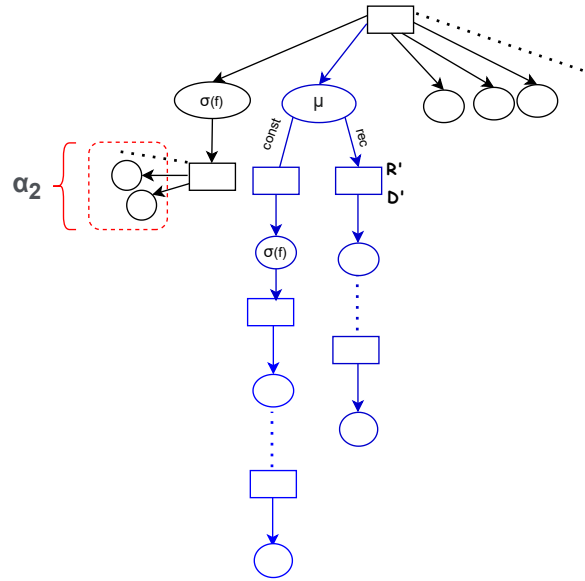
Figure 3.9: Expansion of a RLQDAG by pushing a filter in a fixpoint operation node, with branches added by pf() in blue color.

New branches created by pf() are represented in blue color (on both figures). 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. 3.7: 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 $\mathfrak{R}' = \mathfrak{R} \cup \mathit{filt(f)}$.

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. 3.10 presents the definition of pj() for the two main cases of interest. Again, other cases are defined without structure rearrangement but involving recursive calls to `expand`() in search for further transformation opportunites underneath.

Notice that whenever a join can be pushed within a fixpoint operation node (see Fig. 3.10), 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 refered multiple times (thus explicitly showing the sharing of subterms). Fig. 3.11 illustrates graphically the RLQDAG generated in Fig. 3.10. It shows that the newly created branch (in blue color) extends the set of semantically equivalent terms of the existing equivalence node.

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

- let const $= \gamma$ in

  $\big[\ \big[\mathtt{expand}(\beta) \bowtie \mathtt{expand}(\big[\mu X.\ \mathrm{const} \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}}\big])\big]\ ,$

  $\mu X'.\ \mathtt{expand}(\big[\beta \bowtie [\mathrm{const}]\big]) \cup \big[\mathtt{expand}(\alpha_{\{X/X'\}})\big]_{\mathfrak{D}'}^{\mathfrak{R}'}\ \big]$

  $\textit{when}\quad \textit{type}(\beta) \cap \mathfrak{D} = \emptyset\ \textit{and}\ \textit{type}(\beta)\backslash\textit{type}(\gamma)\ \cap \mathfrak{R} = \emptyset$

- $\big[\mathtt{expand}(\beta) \bowtie \mathtt{expand}(\big[\mu X.\ \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}}\big])\big]\qquad \textit{otherwise}$

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

- let const $= \gamma$ in

  $\big[\ \big[\mathtt{expand}(\beta) \bowtie \mathtt{expand}(\big[\mu X.\ \mathrm{const} \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}},\ \alpha_2\big])\big]\ ,$

  $\mu X'.\ \mathtt{expand}(\big[\beta \bowtie [\mathrm{const}]\big]) \cup \big[\mathtt{expand}(\alpha_{\{X/X'\}})\big]_{\mathfrak{D}'}^{\mathfrak{R}'}\ ,$

  $\big[\mathtt{expand}(\beta) \bowtie \mathtt{expand}(\big[\alpha_2\big])\big]\ \big]$

  $\textit{when}\quad \textit{type}(\beta) \cap \mathfrak{D} = \emptyset\ \textit{and}\ \textit{type}(\beta)\backslash\textit{type}(\gamma)\ \cap \mathfrak{R} = \emptyset$

- $\big[\big[\mathtt{expand}(\beta) \bowtie \mathtt{expand}(\big[\mu X.\ \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}},\ \alpha_2\big])\big]\ ,$

  $\big[\mathtt{expand}(\beta) \bowtie \mathtt{expand}(\big[\alpha_2\big])\big]\ \big]\qquad \textit{otherwise}$

where we define[4] $\mathfrak{D}' = \mathfrak{D} \cup \mathtt{destab}(\beta, X)$ and $\mathfrak{R}' = \mathfrak{R} \cup \mathtt{rigid}(\beta, X)$.

Figure 3.10: Pushing join in an equivalence node containing a fixpoint operation node.

Figure 3.11: Expansion of a RLQDAG by pushing a join in a fixpoint operation node. The initial RLQDAG is in black color, and parts added by pj() are in blue color.

Merging fixpoint operation nodes   The function mf() defined in Fig. 3.12 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. 3.12. 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[5]). New equivalence nodes are created. Fig. 3.13 illustrates the new recursive term produced, using subterm-sharing, after the transformation.

---

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

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

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

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

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

where[6] $\mathfrak{D} = \mathfrak{D}_1 \cup \mathfrak{D}_2$ and $\mathfrak{R} = \mathfrak{R}_1 \cup \mathfrak{R}_2$ and $\alpha_{i\{X_i/X\}}$ denotes α_i in which all occurrences of X_i are replaced by X.

Figure 3.12: Merging fixpoint operation nodes.
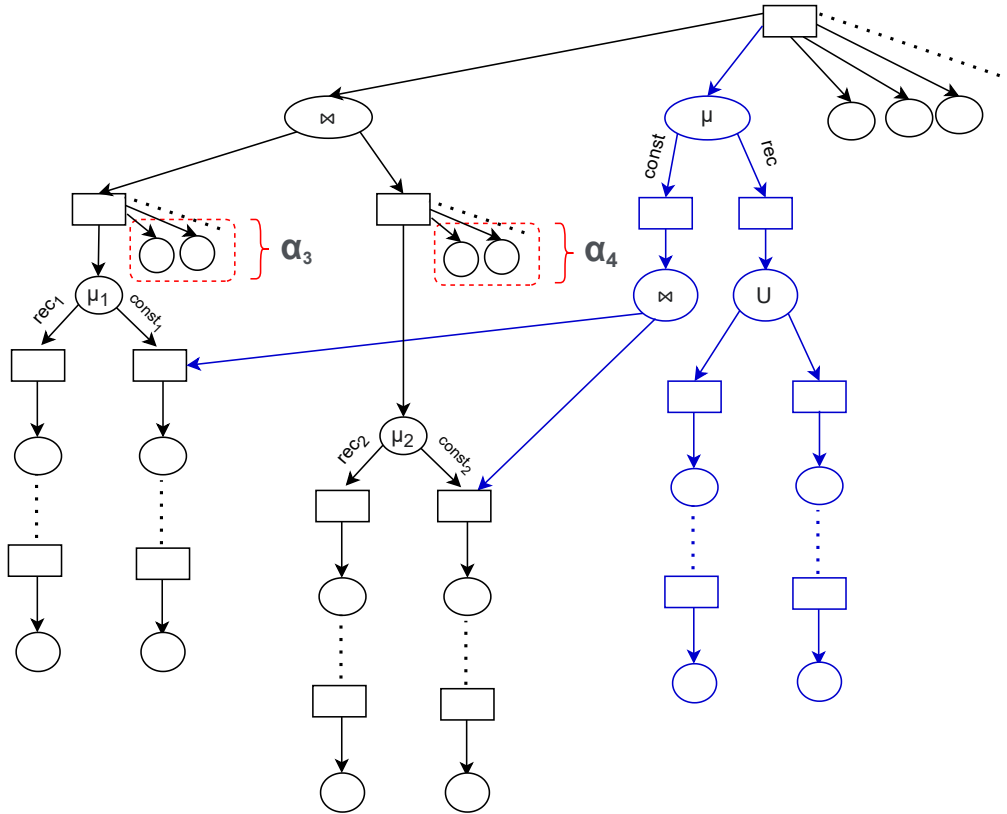
Figure 3.13: RLQDAG structure after merging fixpoint operation nodes.

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

$$\begin{cases} \bullet \quad [\ \mu X'.\ \mathrm{expand}([\widetilde{\pi}_a(\gamma)]) \cup [\mathtt{expand}(\alpha_{\{X/X'\}})]_{\mathfrak{D}'}^{\mathfrak{R}'}\ ] \\[1em] \qquad when\ \ a \notin \mathfrak{R} \\[1em] \bullet \quad [\widetilde{\pi}_a(\mathtt{expand}([\mu X.\ \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}}]))\ ] \qquad otherwise \end{cases}$$

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

$$\begin{cases} \bullet \quad [\ \mu X'.\ \mathrm{expand}([\widetilde{\pi}_a(\gamma)]) \cup [\mathtt{expand}(\alpha_{\{X/X'\}})]_{\mathfrak{D}'}^{\mathfrak{R}'}, \\[1em] \qquad \mathtt{expand}([\widetilde{\pi}_a([\alpha_2])])\ ]\quad when\ \ a \notin \mathfrak{R} \\[1em] \bullet \quad [\ \widetilde{\pi}_a(\mathtt{expand}([\mu X.\ \gamma \cup [\alpha]_{\mathfrak{D}}^{\mathfrak{R}}, \alpha_2])), \\[1em] \qquad \mathtt{expand}([\widetilde{\pi}_a([\alpha_2])])\ ]\qquad otherwise \end{cases}$$

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

Figure 3.14: Pushing antiprojection in an equivalence node containing at least one fixpoint operation node.
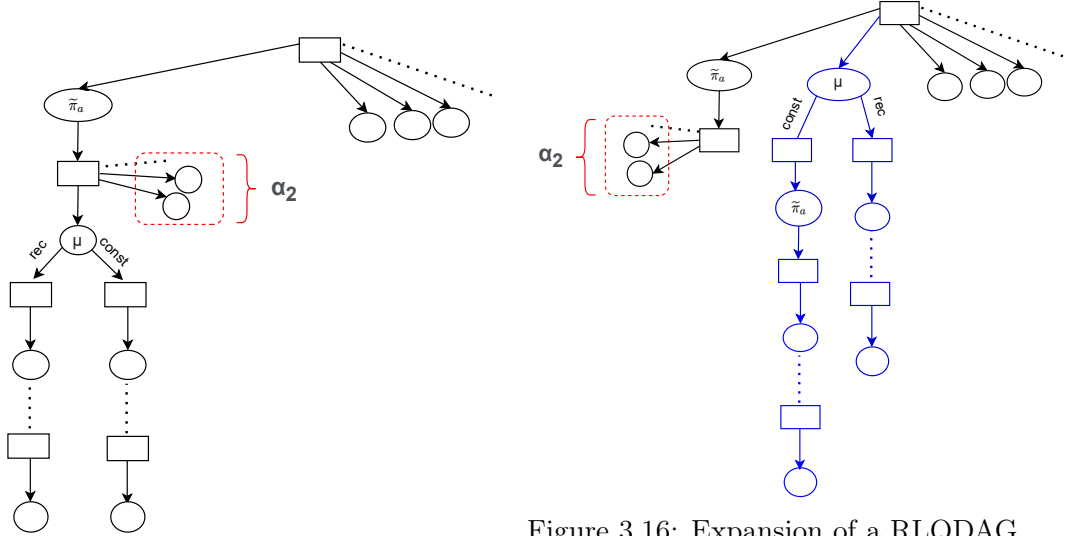
Figure 3.15: Initial RLQDAG before expansion with opportunity to apply pp().

Figure 3.16: Expansion of a RLQDAG by pushing an antiprojection in a fixpoint operation node. The part added by pp() is in blue color.

Pushing antiprojections into fixpoint operation nodes   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. **??**.

The antiprojection is pushed when the criteria allows it, and this results in the

creation of a new term which replaces the initial one in the existing equivalence
node. Whenever the criteria is not satisfied, the initial term is left unchanged,
but traversed in search for more transformation oppotunities.

Fig. 3.15 illustrates graphically the RLQDAG before the transformations.
Fig. 3.16 shows the graphical representation of the RLQDAG obtained by the
transformation of Fig. 3.14.

Pushing antijoins into fixpoint operation nodes   For pushing antijoins
into fixpoints we introduce a function pa() that 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 pushable antijoins have been
pushed in fixpoint operation nodes. pa() is defined in Fig. **??**. Again, Fig. **??**
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. Fig. 3.18 illustrates graphically
the RLQDAG created by the transformations of Fig. 3.18.
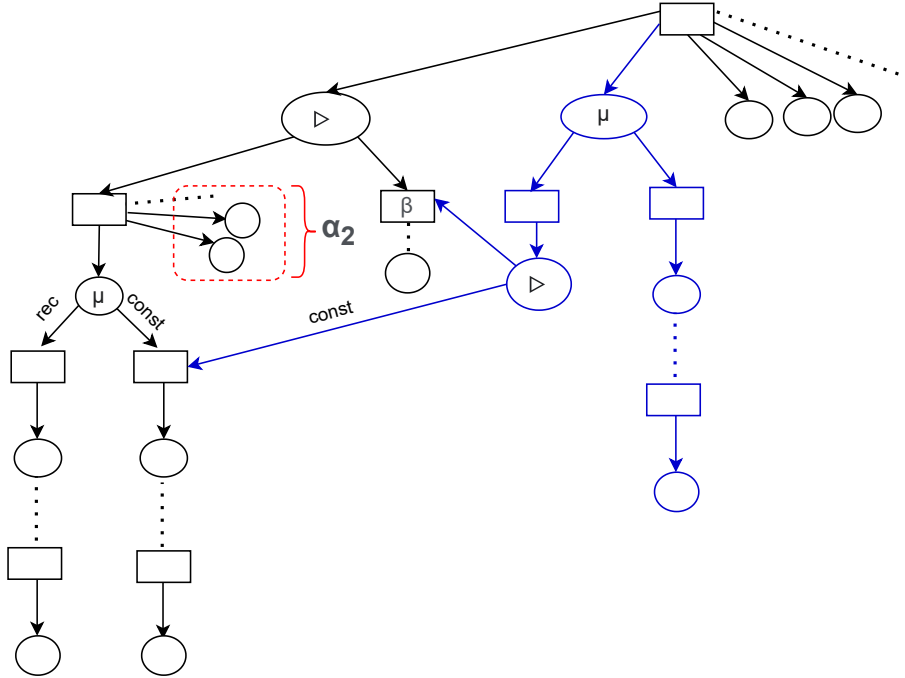
Figure 3.18: Expansion of a RLQDAG by pushing an antijoin in a fixpoint
operation node. The initial RLQDAG is in black color, and parts added by pa()
are in blue color.

Notice that RLQDAG rules can be divided into two groups. The first group
is composed of rewrite rules pj(), pa() and mf(). The application of these rules
always preserve preexisting terms in an equivalence node. These rules can only
perform additions of new operation nodes in an equivalence node[7]. The second

---

[7]One operation node will then be selected using cost estimation heuristics depending on

group is composed of the rules pf() and pp() that perform logical optimizations. The application of these rules can replace terms in an equivalence node by more optimal variants (i.e. they discard logically suboptimal variants).

3.7 The overall expansion algorithm

We can now describe the overall RLQDAG expansion algorithm. We define the function expand() that takes an equivalence node γ and returns the equivalence node γ' which contains all the terms obtained by transformations.

The expand function is simply defined as follows:

$$\texttt{expand}(\lceil d \rceil) = \text{applyAll}(\lceil d \rceil)$$

$$\texttt{expand}(\lceil d, \alpha \rceil) = \text{applyAll}(\lceil d \rceil) \cup \texttt{expand}(\alpha)$$

where applyAll() is in charge of applying all possible transformations on each operation node. This includes applying all rewrite rules defined in Section 3.6 in combination with the more classical ones of relational algebra[8]:

$$\text{applyAll}(\lceil d \rceil) = \text{pf}(\lceil d \rceil) \cup \text{pa}(\lceil d \rceil) \cup \text{pj}(\lceil d \rceil) \cup \text{mf}(\lceil d \rceil)$$
$$\cup \, \text{pp}(\lceil d \rceil) \cup \text{allCodd}(\lceil d \rceil)$$

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

$$\text{allCodd}(\lceil d \rceil) = \text{pfj}(\lceil d \rceil) \cup \text{paj}(\lceil d \rceil) \cup$$
$$\text{jassoc}(\lceil d \rceil) \cup ... \cup \text{dju}(\lceil d \rceil)$$

For instance, pfj() is defined as shown in Fig. 3.19 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 (rewrite rules of Section 2.2.4) are also implemented in a similar way in the RLQDAG.

---

statistics on the cardinalities of the database instance.

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

$\text{pfj}([\sigma_f([\gamma_1 \bowtie \gamma_2, \alpha])]) =$

$$
\begin{cases}
\bullet \quad [\ \text{exp}([\sigma_f(\gamma_1)]) \bowtie \text{exp}(\gamma_2),\ \text{exp}([\sigma_f(\alpha)])\ ] \\
\quad\quad when \ \ filt(f) \subseteq type(\gamma_1) \wedge filt(f) \nsubseteq type(\gamma_2) \\[4pt]
\bullet \quad [\ \text{exp}(\gamma_1) \bowtie \text{exp}([\sigma_f(\gamma_2)]),\ \text{exp}([\sigma_f(\alpha)])\ ] \\
\quad\quad when \ filt(f) \nsubseteq type(\gamma_1) \wedge filt(f) \subseteq type(\gamma_2) \\[4pt]
\bullet \quad [\ \text{exp}([\sigma_f(\gamma_1)]) \bowtie \text{exp}([\sigma_f(\gamma_2)]),\ \text{exp}([\sigma_f(\alpha)])\ ] \\
\quad\quad when \ filt(f) \subseteq type(\gamma_1) \wedge filt(f) \subseteq type(\gamma_2) \\[4pt]
\bullet \quad [\ \sigma_f(\text{exp}([\varphi \bowtie \psi]),\ \text{exp}([\sigma_f(\alpha)]))\ ] \quad otherwise
\end{cases}
$$

Figure 3.19: Pushing a filter in an equivalence node composed of at least one join operation node and other operation nodes ($\text{exp}()$ stands for $\text{expand}()$).

3.8   Correctness and completeness

An important property for the RLQDAG is correctness, which ensures that all structural transformations and terms generated during the expansion process preserve the semantics of the initial term:

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

Proof Sketch. The proof of correctness is done by induction on the structure of α', using a case-by-case analysis of syntactic decompositions (i.e. by considering separately each RLQDAG subcase and the corresponding case of the generalized rewrite rule that applies to it). The complete formal proof is too large to be listed here (with many straightforward subcases). We summarize the main principles as well as useful auxiliary properties. When proving correctness for each RLQDAG rewrite rule, (i) we focus on each newly created recursive term added by the expansion, and (ii) use the consistency hypothesis on the recursive term before transformation to apply the theorems for individual terms of [Jachiet et al., 2020] to each rearrangement of fixpoint operation node. We also need to show that the incremental updates of annotations performed by the RLQDAG rewrite rules preserve consistency. This is done for each update by leveraging the properties that both $\texttt{destab}\ ()$ and $\texttt{rigid}\ ()$ are distributive over union. $\square$

Additional properties of interest ensure that the expansion process does not miss any interesting plan. The following properties state that there is no more unrealized opportunity for transformation in an expanded RLQDAG.

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

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

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

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

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

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

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 $\text{expand}_R()$ leaves no room for such a missed opportunity, thus (iv) leading to a contradiction.

□

Importance of the RLQDAG syntax. Notice that the formal syntax and semantics that we propose (in Sections 3.3 and 3.4) provide a convenient – if not instrumental – means to develop formal proofs[9]. This formalization helped us to detect and fix an intricate completeness issue (caused by a misplaced recursive call) in an earlier version of the RLQDAG.

3.9 Implementation Techniques

In the preceding sections all rewrite rules and functions for applying them have been described in a functional manner. In the prototype implementation, we adopt a more imperative programming style which makes it easier to avoid some redundant computations. In particular, we avoid some redundant calls to the $\text{expand}()$ function when implementing applyAll(). For this purpose, we use the two nested for loop system as shown in Algorithm 1. There we make a call to the imperative variant of the $\text{expand}()$ function after the first for loop in order to call all rewrite rules to be applied (when criteria are satisfied). And then with the

---

[9]This formalization even opens the way for the use of a proof assistant (like Coq) in this context: see the perspectives in Chapter 5.2.

second for loop we examine the other operator to determine which rewrite rule
can be called.

Algorithm 1 ExpandLQDAG

Require: The input is an equivalence node.
Ensure: The output is the expanded equivalence node.
 1: new_set_terms = Set.empty
 2: terms = Set.empty
    for $algOp \in child(eqNode)$ **do**
 3: **case** Filter(fCond, inner_eqNode)**:**
    4:      eqNode = ExpandLQDAG(inner_eqNode)
    5:      terms += Filter(filterCond, eqNode)
          for $inpEq \in input(algOp)$ **do**
 6:      **case** DecomposedMu (α_{const}, α_{rec}, criteria)**:**
       7:          **if** criteria(`destab`(α_{rec}, X), `rigid` (col(fCond), α_{rec}, X)) **then**
       8:             pf(eqNode)
       9:          **end if**
               end case
10:      **case** *another operator***:**
       11:         ...
               end case
          end for
          end case
12: **case** JoinRecursiveTerms(α_1, α_2)**:**
    13:     left = ExpandLQDAG(α_1)
    14:     right = ExpandLQDAG(α_2)
    15:     terms += Join(left, right)
    16:     /* Cross product only when finding fixpoint terms */
    17:     possibilities = crossProduct(keepRecTerms(left), keepRecTerms(right))
          for $inpEq \in possibilities$ **do**
18:      **case** DecomposedMu(α_{const}, α_{rec}), DecomposedMu(α_{const}', α_{rec}') **:**
       19:         new_term =
                   mf(DecomposedMu(α_{const}, α_{rec}), DecomposedMu(α_{const}', α_{rec}'))
       20:         new_set_terms += new_term
               end case
          end for
          end case
21: **case** *another LQDAG***:**
    22:     ...
          end case
    end for
    new_set_terms += terms
    return new_set_terms

Expansion algorithm in an imperative style.   The ExpandLQDAG algo-
rithm takes as input a root equivalence node and generates as output the fully
expanded LQDAG. The fully expanded LQDAG represents the full plan space, i.e.
the set of all plans reachable from the initial term by the application of a given
set of transformation rules. Algorithm 1 is mainly composed of two iterations.
The first iteration (between lines 2 & 3) visits all the algebraic operators that
are direct children of a root equivalence node. For the expansion to be complete

potential recombinations of the initial inner branches need to be explored first hence the recursive call in line 4 (just after first iteration). The second iteration traverses all the inner operators. This amounts to traversing each pair of outer and inner operator. For each such pair any applicable transformation rule is applied. Potential new equivalence nodes on the new terms found are explored by a recursive call on line 10 of the algorithm. When the recursive call happens, not only the new term is created, but at the same time we use a hashing function to check if the term already exists and merge it when possible. Each application of a transformation rule may generate new equivalence nodes or populate existing ones. If the application of a transformation rule results in the creation of an equivalence set that already exists, they are merged. This will be explained more in section 3.10.

We focus on some main cases: pushing filter in a fixpoint and the merging fixpoints. For the first case, we make a call to $pf()$ when criteria are validated. For the second case, we check each two terms, find the cross product between them when both are fixpoint terms and if criteria are validated we make a call to $mf()$ function. To show these cases we use the generalization of 5 rewrite rules concerning fixpoint presented in Section 3.6. When the criteria are verified we can apply the rewrite rules concerning recursion. For each one of the initial terms found in an equivalence node, we make a call to `expand()`. In this case, since we are using the algorithm in an imperative way, the calls to expand fpr the term already in the equivalence node is handled by the call to expand after the first for loop. This way, we remove the redundant call to `expand()` function. When this rule is applied, in the newly created equivalence node we call recursively the `expand()` function. As shown in the adaptation of these rules, the expand function is called recursively in the other plans already present in the equivalence node where the rewrite rule concerning fixpoint is applied. This allows the application of other rewrite rules of relational algebra that might not contain recursion. Using the two for loops and criteria verification, all the equivalence nodes are visited and the totality of rewrite rules are applied.

3.10   Example of unification steps in RLQDAG

When a rewrite rule is applied new equivalence nodes can be created and others can be further populated. When an equivalence node is created, a check verifies whether there already exists an equivalent node (regrouping exactly the same set of subterms). For that purpose, in the implementation, each equivalence node has a unique hash code. This hash code is used to prevent the creation of a new instance of an already existing equivalence node. Instead, it is merged directly with the existing one.

For example, Figure 3.20 illustrates a RLQDAG that corresponds to a query that finds all the Users of a social network that are followed directly or undirectly by a young User of 20 years old. Figure 3.20 shows only the translation in RL form (for the sake of brevity). When the rewrite rule that pushes joins in fixpoints is applied, it first creates a new equivalence operation node (a fixpoint) at the same level as the join operation node being pushed (i.e. under the same equivalence

node). In that particular case, one equivalence node is being populated whilst
other equivalence nodes are being created. Each one of them is checked for
pre-existence at the moment of creation to avoid duplicates. In this example we
focus on the creation of a whole branch for the sake of clarity and simplification.
In RLQDAG, the application of rewrite rules occurs for each equivalence node
separately. The first branch created is the one illustrated in Figure 3.21a. The
join operator is pushed in the constant part of the fixpoint operator and that is
the first branch to be created. The term in green (on the left side) is the one
being created, and the green one on the right side is the pre-existing one. Both
subparts are detected as equal (based on hash codes) and are thus merged. This
results in two operation nodes at upper level pointing to the same equivalence
node. This is depicted in Figure 3.21b. Then, in a subsequent step, the other
operand of join at the same level is created. This is shown in blue on the left
side of Figure 3.22a. Again, as in the previous example, a duplicate subpart is
identified and merged in a single one. This is illustrated in Figure 3.22b. In a
later step, a similar situation happens with the creation of the yellow branch, as
illustrated in Figure 3.23a which is also unified with its preexisting version. In the
end, all the nodes created by this rewrite rule application are expanded and the
final RLQDAG is the one shown in Figure 3.23b. As we can observe, the cascade
of unifications results in the creation of only the red part. More generally, plan
enumeration is conducted with all rewrite rules in a recursive top-down manner.
We showed only some "macro" steps on RLQDAG expansion. This means that
other equivalence nodes in the deeper levels are visited and expanded as well.
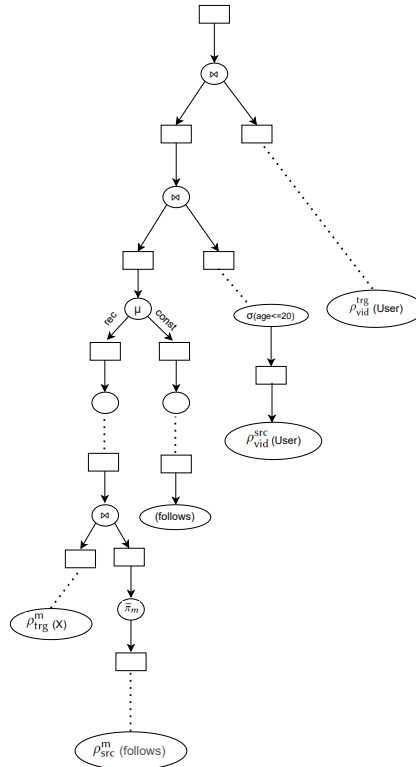
Figure 3.20: RLQDAG example of a graph query.

(a) Branch creation.

(b) Unification of constant parts of recursive term.
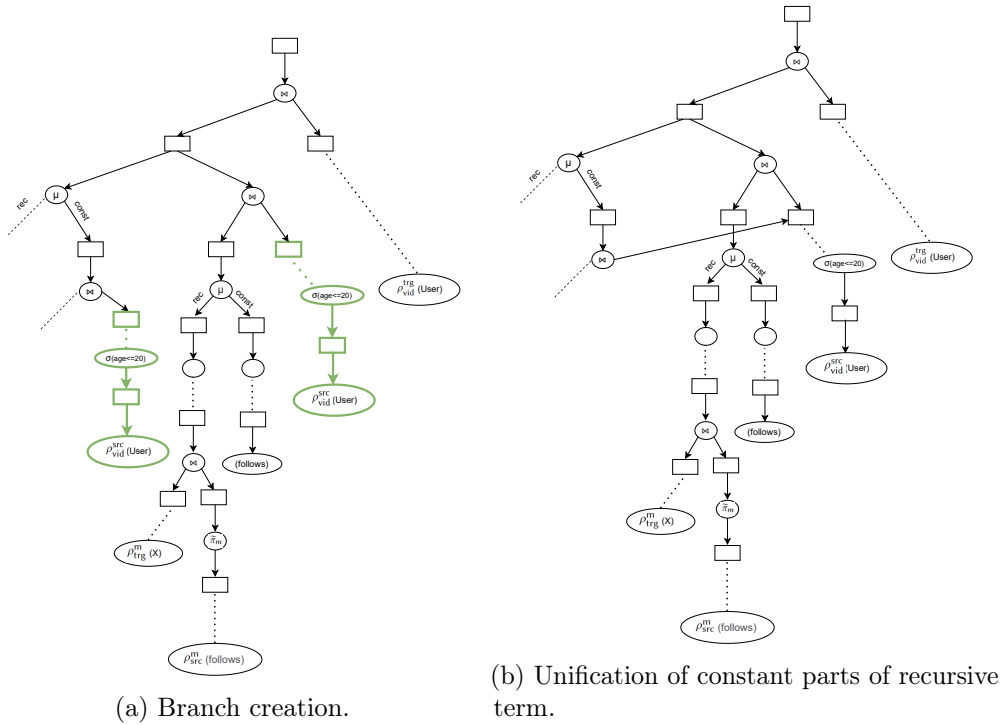
Figure 3.21: Creation and unification of constant parts of the recursive term.

(a) Another tree branch creation.

(b) Unification of branches.

Figure 3.22: Creation and unification of branches on the recursive term.

(a) Recursive branch creation.

(b) Final unification.

Figure 3.23: Recursive branch creation and final unification.

3.11 Non-regular queries

The RLQDAG can not only express regular expressions, but non-regular ones as well. One example of a non-regular term is $R_1^n R_2^n$, where R_1 and R_2 are relations.

$R_1^n R_2^n$ written as RLQDAG

$$\mu(X.\ \left[\widetilde{\pi}_{\mathsf{m}}\left(\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_1\right]\right)\right]\bowtie\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_2\right]\right)\right]\right)\right]$$

$$\cup\ \left[\widetilde{\pi}_{\mathsf{m}}\left(\widetilde{\pi}_{\mathsf{n}}\left(\left[\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_1\right]\right)\right]\bowtie\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[\rho_{\mathsf{src}}^{\mathsf{n}}\left(X\right)\right]\right)\right]\right)\bowtie\left[\rho_{\mathsf{src}}^{\mathsf{n}}\left(\left[R_2\right]\right)\right]\right)\right]_{\{\mathsf{src,trg}\}}^{\{\mathsf{src,trg,m,n}\}}\right)$$

Pushing a filter in $R_1^n R_2^n$

Generalized rewrite rules apply to any RLQDAG terms, in particular to non-regular terms.

Below we show an example where a filter is pushed in the previous RLQDAG. The filter is pushed in the constant part of the recursive term when criteria are satisfied. This new term is added to the existing equivalence node. We explain this as follows: Let's suppose an equivalence node that contains term t below:

$$\left[\sigma_f(\left[\mu(X.\ \left[\widetilde{\pi}_{\mathsf{m}}\left(\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_1\right]\right)\right]\bowtie\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_2\right]\right)\right]\right)\right]\right.$$

$$\cup\ \left[\widetilde{\pi}_{\mathsf{m}}\left(\widetilde{\pi}_{\mathsf{n}}\left(\left[\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_1\right]\right)\right]\bowtie\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[\rho_{\mathsf{src}}^{\mathsf{m}}\left(X\right)\right]\right)\right]\right)\bowtie\left[\rho_{\mathsf{src}}^{\mathsf{n}}\left(\left[R_2\right]\right)\right]\right)\right]_{\{\mathsf{src,trg}\}}^{\{\mathsf{src,trg,m,n}\}}\right)\right])\right]$$

This equivalence node can be written as $\left[t\right]$. When the rewrite rule of pushing a filter in a fixpoint is applied, the equivalence node is expanded with term t', written below:

$$\left[\mu(X.\ \left[\sigma_f(\left[\widetilde{\pi}_{\mathsf{m}}\left(\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_1\right]\right)\right]\bowtie\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_2\right]\right)\right]\right)\right]\ )\ \right]\right.$$

$$\cup\ \left[\widetilde{\pi}_{\mathsf{m}}\left(\widetilde{\pi}_{\mathsf{n}}\left(\left[\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[R_1\right]\right)\right]\bowtie\left[\rho_{\mathsf{trg}}^{\mathsf{m}}\left(\left[\rho_{\mathsf{src}}^{\mathsf{m}}\left(X\right)\right]\right)\right]\right)\bowtie\left[\rho_{\mathsf{src}}^{\mathsf{n}}\left(\left[R_2\right]\right)\right]\right)\right]_{\{\mathsf{src,trg}\}}^{\{\mathsf{src,trg,m,n}\}}\right)\right]$$

Now the equivalence node contains both terms $\left[t, t'\right]$.

In the next Chapter 4, we describe an application to graph queries in which regular RLQDAG terms are automatically generated from a higher-level graph query language.

Chapter 4

Application to property graph queries

This Chapter presents an application of the RLQDAG (introduced in Chapter 3) for extracting information from property graphs. It presents the experimental assesment of all concepts and algorithmic techniques proposed in Chapter 3, using a complete RLQDAG prototype implementation. We first present a recursive query language fragment (UCRPQ$_{\text{PG}}$) adapted for property graphs. We propose a direct translation of queries in this fragment into the RLQDAG. We then conduct and report on experiments using real and synthetic property graphs.

4.1 Property graph representation in RA data model

We first explain how property graphs can be represented so as to be exposed to relational queries. In the property graph data model, nodes and edges are labeled and they can also carry a list of properties with values, as described in Section 1.1.2 of Chapter 1. In relational algebra, a property graph can be represented as a set of relations, with one relation per edge type and one relation per node type. Each relation encodes the specific properties of each node (or edge respectively) using one column per property. Figure 4.1 illustrates an example of a possible representation of a social network property graph. The table on the left of Figure 4.1 encodes nodes where each node corresponds to a person. We can see that this table has a column `vid` and several other columns that can be subject to filtering based on values. The table on the right of Figure 4.1 illustrates an example of a relation encoding edges in this social network property graph, where we can see columns `src` and `trg` along with other columns giving more information on the nature of the relation that connects two nodes.

vid	name	lastname	age	...
101	Bob	Dylan	35	
102	Alice	Hawke	30	
103	John	Doe	26	
...				

src	trg	since	suggested_from	...
101	202	2015	family	
102	203	2019	friend	
103	201	2017	cousin	
...				

Figure 4.1: Possible relational encoding of a social network property graph, with vertices on the left and edges on the right.

Figure 4.2 illustrates an example of a social graph schema. There is one relation for each type of nodes (User, Forum, Tag) and one relation for each type of edges (follows, member_of, has_interest, has_tag). Each of these relations has a list of properties with respective values.
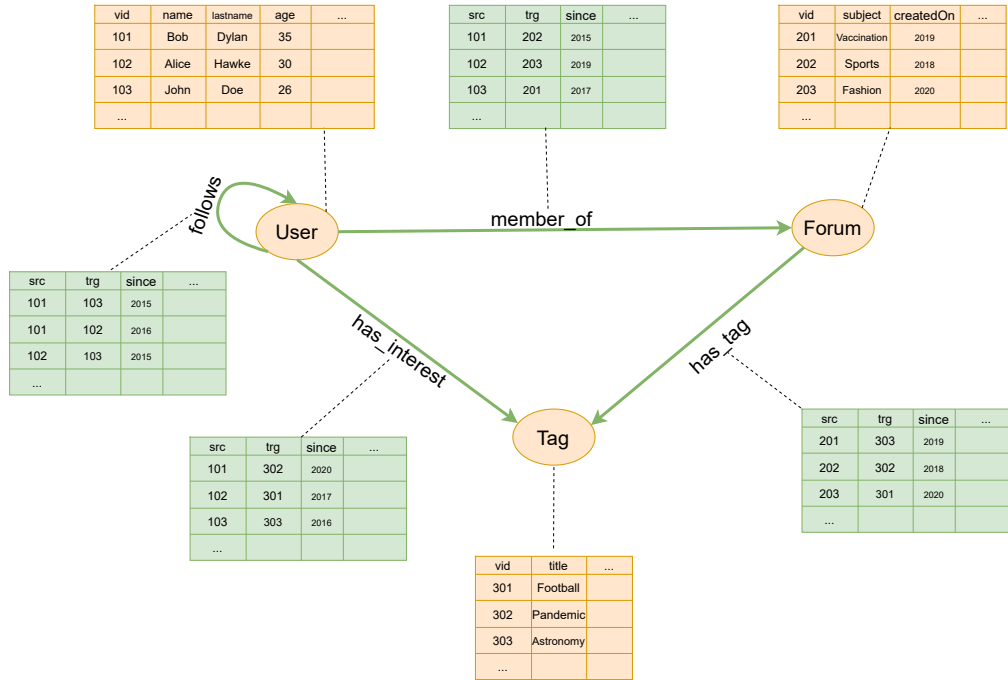
Figure 4.2: Representation of a social network property graph.

This representation comes with several assumptions to ensure consistency when exposed to relational queries. In particular, nodes must be uniquely identified (the vid column) and those identifiers must be disjoint between node types. For each edge type, the corresponding relation contains at least the source and target nodes (which are foreign keys to node's vids). A knowledge graph is simply a particular case of a property graph with no particular property beyond source and target vids in edge relations.

4.2 A recursive query language fragment suited for property graphs

4.2.1 Idea

To support recursive queries on property graphs, we consider a query language fragment for property graphs, named UCRPQ$_{PG}$, that we propose. This fragment is inspired from the recursive query language fragments reviewed in Chapter 1: it is a variant of UCRPQs which is slightly generalized to offer specific filtering capabilities suited for the rich property graph data model. In the knowledge graph data model used earlier in μ-RA [Jachiet et al., 2020], it was possible to filter based on the values found in the label column of the table. For example, in the knowledge graph representation, in order to consider only the edge "livesIn", we had to filter the value "livesIn" on label column (label="livesIn"). Using the property graph representation, we consider separate tables for each node and edge type and we want to offer the ability to filter based on (key:value) pairs for each property. Given the above example of the graph in Figure 4.2, when finding users aged 30 years old, we would like to filter directly on column `age=30`. UCRPQ$_{PG}$ queries make this possible. For instance, if one wants to express a UCRPQ$_{PG}$ query that considers the members who joined since 2015, it suffices to filter on column `since=2015` of table `member_of` and consider this as the query along with the nodes and other requirements if any. One query example is the query that finds the totality of users followed by Bob (directly or indirectly) since 2013 and that are also members of a given Forum y since 2020. We can formulate this as a UCRPQ$_{PG}$ query, as follows:

$$?x, ?y \leftarrow \ ?b : \texttt{User[name:Bob] follows[since = 2013]+} \ ?x : \texttt{User},$$
$$?x : \texttt{User member\_of[since = 2020]} \ ?y : \texttt{Forum}$$

In this formulation, expressions between brackets denote filters on nodes and edges and it corresponds to the example of Figure 4.2.

In general, UCRPQ$_{PG}$ queries typically contain many more filters (and joins) than queries on knowledge graphs, as they are specifically suited for the richer property graph data model. When these additional filtering capabilities are combined with recursion, we will see that this can create significantly bigger plan spaces during the plan exploration phase. As such, UCRPQ$_{PG}$ queries represent a particularly relevant and challenging baseline for an experimental assessment of the RLQDAG introduced in Chapter 3. We first introduce an abstract syntax for $UCRPQ_{PG}$ queries and we propose direct formal translations of $UCRPQ_{PG}$ queries into the RLQDAG representation.

4.2.2 Syntax of UCRPQ$_{PG}$

We define UCRPQ$_{PG}$ queries as the set of expressions that can be formed using the abstract syntax shown in Figure 4.3. One core component is the syntax of regular expressions r over edges. The base case for a regular expression r is an edge label which can also be further refined using a set of keys filtered with

constant values. Nodes can also be filtered similarly. Nodes can be named by
introducing fresh variables names (?var), that might be reused in the query to
express sophisticated connections between nodes.

```
      Node
node_pattern  ::=

                  ?var  :  nodeLabel map
              |   ?var  :  label_list
              |   *
```

```
   Relation
 rel_pattern  ::=  r
          r   ::=                                          reg. exp. over edges
                  edgeLabel map?
              |   r r                                      concatenation
              |   r | r                                    alternative
              |   -r                                       reverse
              |   r+                                       transitive closure
```

```
   Properties
       map    ::=  {prop_list}
    prop_list ::=  k:v | k:v, prop_list
```

```
Conjunct (c)
          c   ::=

                  node_pattern  rel_pattern  node_pattern
              |   c, node_pattern  rel_pattern  node_pattern
```

Figure 4.3: Syntax of a query language on property graphs.

Formally, a CRPQ_{PG} query is of the form $H \leftarrow C$, where the query head H
is a non-empty set of vertex variables to be extracted by the query, and C is
a conjunction that describes how those vertex variables are connected to other
vertex variables or to constants. More formally:

- H is a sequence of variables of the form $(z_1, ..., z_m)$ which is not empty
  (we do not consider boolean queries). These variables are taken from the
  variables specified in each one of the nodes of a conjunction c_i.

- C is a non-empty conjunction $c_1, c_2..., c_n$ in which each conjunct c_i has the
  form

$$x : \text{vartype1 map?} \quad r \quad y : \text{vartype2 map?}$$

where:

- x (and respectively y) is a variable

- vartype is the name of the label which decides the $R_{:\ell}$ used in each node

- r follows the syntax defined previously for expressing a regular expression over edges in the property graph.

UCRPQ$_{\text{PG}}$ queries extend CRPQ$_{\text{PG}}$ with union at top level. They have the syntax $H \leftarrow C_1 \cup ... \cup C_n$ in which each disjunct C_i is a conjunction as defined previously.

4.2.3 Translation into RLQDAG

UCRPQ$_{\text{PG}}$ queries can be supported in the RLQDAG. We show how UCRPQ$_{\text{PG}}$ queries can be directly translated into the RLQDAG, based on the assumptions introduced in Section 4.2.3.1 and that we specify more formally below.

4.2.3.1 Assumptions

As mentioned previously, a property graph is represented as a set of relational tables (in contrary to a knowledge graph that is represented by a single table). The formal translation of UCRPQ$_{\text{PG}}$ relies on a few assumptions on the way the property graph is represented as a set of relations. It assumes that the following hypotheses are satisfied:

- there exists a μ-RA relation $R_{:\ell}$ for each label $:\ell$ in label_list

- there exists a μ-RA relation R_t for each label t in type_list

- each relation $R_{:\ell}$ contains one column named vid (at least)

- each relation R_t contains one column src and one column trg (at least)

- $\mathcal{V}$ is a set of column names (that start with a leading '?') which are reserved for the translation. $\mathcal{V}$ is assumed to be disjoint from the set of column names used in any of the relations R_t and $R_{:\ell}$.

- each relation $R_{:\ell}$ has distinct vid values. There is no intersection of vid values between different R_i. In other terms, for any two R_i and R_j with $i \neq j$, $\text{vid}_{R_i} \cap \text{vid}_{R_j} = \emptyset$

4.2.3.2 Translation functions

In order to translate UCRPQ$_{\text{PG}}$ queries into RLQDAG, we use a set of formal translation functions defined below. First of all, we define the function $\text{tr}_r(\cdot)$ that takes as input an expression r over edges (as defined in Figure 4.3), and builds the corresponding RLQDAG:

$$\mathtt{tr}_r(\mathrm{edgeLabel}) = \big[\mathrm{edgeLabel}\big]$$

$$\mathtt{tr}_r(\mathrm{edgeLabel\ map}) = g\big(\big[\mathrm{edgeLabel}\big]\big) \quad \text{with } g = \mathtt{tr}_{\mathsf{map}}(\mathsf{map})$$

$$\mathtt{tr}_r(r\ r') = \big[\mathtt{tr}_r(r) \bowtie \mathtt{tr}_r(r')\big]$$

$$\mathtt{tr}_r(r \mid r') = \big[\mathtt{tr}_r(r) \cup \mathtt{tr}_r(r')\big]$$

$$\mathtt{tr}_r(\text{-}r) = \big[\rho_{\mathsf{m}}^{\mathsf{trg}}\big(\big[\rho_{\mathsf{trg}}^{\mathsf{src}}\big(\big[\rho_{\mathsf{src}}^{\mathsf{m}}\left(\mathtt{tr}_r(r)\right)\big]\big)\big]\big)\big]$$

$$\mathtt{tr}_r(r\text{+}) = \ \mathtt{let\ rel} = \mathtt{tr}_r(r) \ \mathtt{in}$$

$$\big[\ \mu X.\ \mathtt{rel} \cup \big[\tilde{\pi}_{\mathsf{m}}(\big[\rho_{\mathsf{trg}}^{\mathsf{m}}(\big[\mathtt{rel}\big])\big] \bowtie \big[\rho_{\mathsf{src}}^{\mathsf{m}}(\big[X\big])\big])\big]_{\{\mathtt{src}\}}^{\{\mathtt{src},\mathtt{trg},\mathtt{m}\}} \ ,$$

$$\mu X.\ \mathtt{rel} \cup \{\tilde{\pi}_{\mathsf{m}}(\big[\rho_{\mathsf{src}}^{\mathsf{m}}(\big[\mathtt{rel}\big])\big] \bowtie \big[\rho_{\mathsf{trg}}^{\mathsf{m}}(\big[X\big])\big])\}_{\{\mathtt{trg}\}}^{\{\mathtt{src},\mathtt{trg},\mathtt{m}\}} \ \big]$$

Two initial translations of transitive closure. $\mathtt{tr}_r(r\text{+})$ returns a RLQDAG which contains two different forms for translating the transitive closure R^+ of a basic relation R. Intuitively, these two forms correspond to two possible directions with which a fixpoint term can compute R^+. The first form amounts to computing R+ in the right to left direction (RL), whereas the second form computes the same relation in the reverse direction, left to right (LR).

It is useful to keep track of both forms as transformation rules apply differently on each form. For example: a filter on the *src* column can be pushed in the form that navigates from left to right, but not on the other one. Similarly a filter on the *trg* column can be pushed only in the form that navigates to the left.

The two forms are semantically equivalent (they eventually return the same set of results). For this reason, they are thus grouped under the same equivalence node in the generated RLQDAG. The above translation shows explicitly the annotations carried by annotated equivalence nodes, that are used by the RLQDAG rewriting rules presented in Chapter 3.

Filtering of properties based on key values.   Filtering based on properties is applied both in nodes and in edges. For each label_list in the case of nodes and for each type_list in the case of edges there is a specific list of properties. The idea is the same, in both cases, since the filtering is applied in the list of properties. That is why the following translations are valid for both nodes and edges. The list in the translation example is a label_list in case of nodes, or a type_list in case of edges.

The function $\mathtt{tr}_{\mathsf{map}}(\mathsf{map})$ returns a filtering function:

$$\mathtt{tr}_{\mathsf{map}}(\{\mathsf{prop\_list}\}) = \mathtt{tr}_{\mathsf{prop\_list}}(\mathsf{prop\_list})$$

$$\mathtt{tr}_{\mathsf{prop\_list}}(k\!:\!v) = \lambda x.\sigma_{k=v}\big(\big[x\big]\big)$$

$$\mathtt{tr}_{\mathsf{prop\_list}}(k\!:\!v,\ \mathsf{prop\_list}) = \lambda x.\sigma_{k=v}(\mathtt{tr}_{\mathsf{prop\_list}}(\mathsf{prop\_list})\big(\big[x\big]\big))$$

Pattern composition in CRPQ$_{\text{PG}}$. The translation of pattern composition shows the connection between the nodes and the relation pattern. In the following translations we use var_{i} for denoting variables used on nodes and these ones are projected on (shown) at the end. Below we show all possibilities, even when a node is not specified.

$$
\begin{aligned}
\text{tr}_{\text{pattern}}(\text{node\_pattern rel\_pattern node\_pattern}) = \{&\text{tr}_n(\text{node\_pattern}) \\
&\bowtie_{?\text{var}_1} \rho_{\text{src}}^{?\text{var}_1}(\rho_{\text{trg}}^{?\text{var}_2}(\varphi)) \\
&\bowtie_{?\text{var}_2} \text{tr}_n(\text{node\_pattern}) \\
&\mid \varphi \in \text{tr}_r(\text{rel\_pattern})\} \\
\text{tr}_{\text{pattern}}(\text{node\_pattern rel\_pattern} *) = &\text{tr}_n(\text{node\_pattern}) \\
&\bowtie_{?\text{var}_1} \rho_{\text{src}}^{?\text{var}_1}(\text{tr}_r(\text{rel\_pattern})) \\
\text{tr}_{\text{pattern}}(* \text{ rel\_pattern node\_pattern}) = &\rho_{\text{trg}}^{?\text{var}_2}(\text{tr}_r(\text{rel\_pattern})) \\
&\bowtie_{?\text{var}_2} \text{tr}_n(\text{node\_pattern}) \\
\text{tr}_{\text{pattern}}(* \text{ rel\_pattern} *) = &\text{tr}_r(\text{rel\_pattern})
\end{aligned}
$$

The initial RLQDAGis created thanks to these translation functions. Then we can apply the generalized rewrite rules presented in Chapter 3 in order to efficiently explore the plan space for a given query.

4.2.4 Example of RLQDAG generated from UCRPQ$_{\text{PG}}$

Let us consider the query we already presented in the preceding section:

$$?u, ?f \leftarrow ?b : \texttt{User[name:Bob] follows[since = 2013]+ } ?u : \texttt{User},$$
$$?u : \texttt{User member\_of[since = 2020] } ?f : \texttt{Forum}$$

This UCRPQ$_{\text{PG}}$ query is meant to be evaluated on the graph presented in Figure 4.2. The translation generates the RLQDAG term shown in Figure 4.4.

```
let rel = σ_since=2013([follows]) in
let cst = ρ_src^b([ρ_trg^u(rel)]) in
let user_u = ρ_vid^u([User]) in
[ [    [ρ_vid^b([σ_name=Bob([User])])]  ⋈
    [ μX. cst ∪ [π̃_m([ρ_trg^m(rel)] ⋈ [ρ_src^m([X])])]_{src}^{src,trg,m} ,
    μX. cst ∪ [π̃_m([ρ_trg^m(rel)] ⋈ [ρ_trg^m([X])])]_{trg}^{src,trg,m} ] ]
⋈ user_u ]
         ⋈
[ [ user_u   ⋈  [ρ_src^u([ρ_trg^f([σ_since=2020([member_of])])])] ]  ⋈  [ρ_vid^f([Forum])] ] ] ]
```

Figure 4.4: RLQDAG term obtained by the translation of a sample UCRPQ$_{\text{PG}}$.

The red part corresponds to the navigation from right to left RL in a recursive term, whereas the blue one illustrates the one from left to right LR. In the above translation we also show the generated annotations carried by annotated equivalence nodes. This RLQDAG term is also depicted graphically in Figure 4.5. The annotations for LR are $\mathfrak{D}$ and $\mathfrak{R}$ in blue. $\mathfrak{D}$ contains $\{\texttt{src}\}$ whilst $\mathfrak{R}$ is $\{\texttt{src},\texttt{trg},\texttt{m}\}$. The annotations for RL are $\mathfrak{D}_1$ and $\mathfrak{R}_1$ in red. $\mathfrak{D}_1$ is $\{\texttt{src}\}$ whilst $\mathfrak{R}_1$ is $\{\texttt{src},\texttt{trg},\texttt{m}\}$.

Figure 4.5: Graphical representation of the RLQDAG term obtained from the translation.

Pushing a Join in RLQDAG.   In Figure 4.6 the rewrite rule that pushes a join into a fixpoint operation node is applied. Its graphical illustration is shown in Figure 4.7. The annotations are different for RL and LR, hence the terms allowed to be pushed in each one of the forms will be different. With the rewrite rule application, the new terms formed trigger an incremental update of annotations from the previous annotations of recursive terms. For the sake of simplicity, in both Figure 4.6 and Figure 4.7 only the new terms created are shown, the terms initial present in the equivalence node are not shown.

```
let rel = σ_since=2013([follows]) in
let cst = ρ_src^b ([ρ_trg^u (rel)]) in
let user_u = ρ_vid^u ([User]) in
let user_b = ρ_vid^b ([σ_name=Bob([User])]) in
```

$$\Big[\; \Big[\texttt{user\_b} \; \bowtie$$

$$\big[\mu X. \big[\texttt{user\_u} \bowtie \texttt{cst} \big] \; \cup \; \big[\tilde{\pi}_{\texttt{m}}(\big[\rho_{\texttt{trg}}^{\texttt{m}}(\texttt{rel})\big] \bowtie \big[\rho_{\texttt{src}}^{\texttt{m}}([X])\big])\big]_{\{\texttt{src}\}}^{\{\texttt{src,trg,m,vid,u}\}} \big] \; \big],$$

$$\big[\; \big[\mu X. \big[\texttt{user\_b} \bowtie \texttt{cst} \big] \; \cup \; \big[\tilde{\pi}_{\texttt{m}}(\big[\rho_{\texttt{trg}}^{\texttt{m}}(\texttt{rel})\big] \bowtie \big[\rho_{\texttt{trg}}^{\texttt{m}}([X])\big])\big]_{\{\texttt{trg}\}}^{\{\texttt{src,trg,m,vid,b}\}} \big]$$

$$\bowtie \; \texttt{user\_u} \big] \; \big]$$

$$\bowtie$$

$$\Big[\; \big[\texttt{user\_u} \quad \bowtie \; \big[\rho_{\texttt{src}}^{\texttt{u}} \big([\rho_{\texttt{trg}}^{\texttt{f}} \big([\sigma_{\texttt{since=2020}}([\texttt{member\_of}])]\big)]\big)\big] \; \big] \quad \bowtie \; \big[\rho_{\texttt{vid}}^{\texttt{f}} \big([\texttt{Forum}])\big) \big] \; \big] \; \big]$$

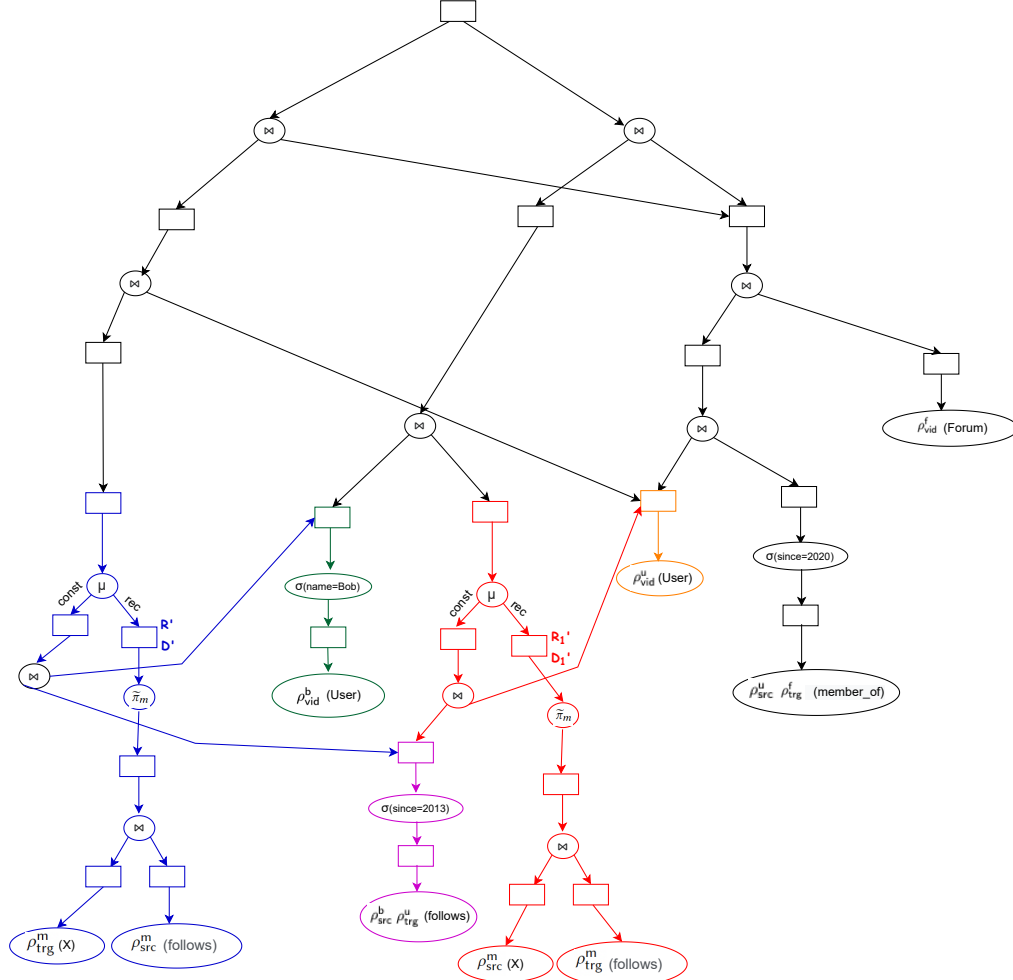Figure 4.6: RLQDAG term obtained after pushing a join into a fixpoint operation
node.

Figure 4.7: RLQDAG term graphical representation after join is pushed into a
fixpoint operation node.

In LR, the term in green corresponding to the User named Bob is pushed in the fixpoint. We will consider $\mathfrak{D}_\mathfrak{b}$ the `destab` of `user_b`, its `rigid` is considered as $\mathfrak{R}_\mathfrak{b}$. Based on derivations of `destab`, $\mathfrak{D}' = \mathfrak{D} \cup \mathfrak{D}_\mathfrak{b}$. $\mathfrak{D}_\mathfrak{b}$ is *emptyset* (`destab` is the one of User which is considered in this case), hence $\mathfrak{D}'$ is still `trg`. Based on `rigid`, $\mathfrak{R}' = \mathfrak{R} \cup \mathfrak{R}_\mathfrak{b}$. $\mathfrak{R}_\mathfrak{b} = \mathtt{vid}, \mathtt{b}$, hence $\mathfrak{R}' = \{\mathtt{src}, \mathtt{trg}, \mathtt{m}, \mathtt{vid}, \mathtt{b}\}$. The same calculations are done for RL, but there is `user_u` being pushed so $\mathfrak{D}'_1$ and $\mathfrak{R}'_1$ are updated in accordance.

4.3 Experimental setup for the RLQDAG

In order to assess the efficiency of the RLQDAG approach in practice, we implemented a prototype of the RLQDAG in the μ-RA system [Jachiet et al., 2020], which is, as described in Section 2.7 of Chapter 2, one of the most advanced relational-based system for recursive query optimization; and the system with the richest plan spaces for recursive terms, that are explored using a dynamic programming strategy [Jachiet et al., 2020]. We compare the performance of our RLQDAG prototype implementation with the performance of the unmodified μ-RA system.

4.3.1 Datasets

In order to assess the efficiency of the RLQDAG in practice, we consider recursive graph queries formulated against various real and generated datasets:

- Yago [YAGO, 2019], a knowledge graph containing $62,643,951$ edges, $42,832,856$ nodes and 83 predicates.

- Bahamas Leaks [Bahamas-Leaks, 2017], a property graph that provides names of directors and owners of Bahamian companies, trusts and foundations registered between 1990 and 2016, together with their connections. It contains $202,242$ nodes and $249,190$ edges.

- Airbnb [Airbnb, 2022], a property graph with $24,840$ nodes and 14000 edges.

- LDBC social network [Boncz, 2013], a property graph with $908,224$ nodes and $1,960,654$ edges.

4.3.2 Queries

Queries are chosen to reflect a variety of patterns commonly found in practice when querying property graphs. Property graph queries are written using UCRPQ_{PG}. Queries for Yago are mainly third-party regular path queries already considered in earlier papers in the litterature [Jachiet et al., 2020, Abul-Basher et al., 2017b, Yakovets et al., 2015b, Gubichev et al., 2013], 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 [Sharma et al., 2021]. We added more queries formulated over the Bahamas and LDBC datasets. Queries

and datasets used in experiments are available at [Tyrex-repository, 2022] and shown as well as in the appendix of this thesis.

We systematically compare results obtained with the RLQDAG with those obtained using the state-of-the-art μ-RA system implementation [Jachiet et al., 2020].

Since RLQDAG is focused on recursive queries, we want to show different ways of application of recursion. For this reason we present a classification of recursive queries in seven classes, inspired from [Chlyah et al., 2022].

- $\mathcal{C}_0$: recursion-free queries (for which only classical rewrite rules for non-recursive terms will be triggered).

- $\mathcal{C}_1$: Single recursion, e.g. $?x, ?y \leftarrow ?x\ a+\ ?y$

- $\mathcal{C}_2$: Filter to the right of a recursion, e.g. $?x \leftarrow ?x\ a+\ C$. In the case of property graphs (our representation), C can be expressed as: $x : colName[prop1 = value1]$.

- $\mathcal{C}_3$: Filter to the left of a recursion, e.g. $?x \leftarrow C\ a+\ ?x$.

- $\mathcal{C}_4$: Concatenation of a non recursive term to the right of a recursion, e.g. $?x, ?y \leftarrow ?x\ a+/b\ ?y$.

- $\mathcal{C}_5$: Concatenation of a non recursive term to the left of a recursion, e.g. $?x, ?y \leftarrow ?x\ b/a+\ ?y$.

- $\mathcal{C}_6$: Concatenation of recursions, e.g. $?x, ?y \leftarrow ?x\ a+/b+\ ?y$.

The classification for queries used in experiments is shown in Table 4.1. All queries over the different datasets are chosen such that they belong to more than one category in this classification. Queries cover all classification possibilities.

4.3.3 Hardware setup

The experiments were conducted using a machine with a 2,8 GHz Quad-Core Intel Core i7; 16GB memory DDR3. The version of postgres used is 10.5, run on Docker 19.03.

4.4 Experimental Results

In this section we report on the results obtained in two phases: (i) the plan enumeration phase of RLQDAG, during which we assess the efficiency of the exploration of the space of recursive plans; (ii) the query evaluation phase where we assess to which extent the efficiency of enumeration is beneficial when running the best estimated term from the explored plan space.

Query	$\mathcal{C}_0$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$	$\mathcal{C}_4$	$\mathcal{C}_5$	$\mathcal{C}_6$	Query	$\mathcal{C}_0$	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$	$\mathcal{C}_4$	$\mathcal{C}_5$	$\mathcal{C}_6$
$\mathcal{Q}1$	+		+			+	+	$\mathcal{Q}28$	+			+		+	
$\mathcal{Q}2$	+		+			+	+	$\mathcal{Q}29$	+	+	+				
$\mathcal{Q}3$	+		+			+	+	$\mathcal{Q}30$	+	+	+	+			
$\mathcal{Q}4$	+		+			+	+	$\mathcal{Q}31$	+	+		+			
$\mathcal{Q}5$	+		+			+	+	$\mathcal{Q}32$	+	+		+			
$\mathcal{Q}6$	+		+			+	+	$\mathcal{Q}33$	+	+					
$\mathcal{Q}7$	+		+			+	+	$\mathcal{Q}34$	+			+			+
$\mathcal{Q}8$	+		+				+	$\mathcal{Q}35$	+	+	+	+			
$\mathcal{Q}9$	+	+						$\mathcal{Q}36$	+	+		+			
$\mathcal{Q}10$	+			+	+	+		$\mathcal{Q}37$	+	+		+			
$\mathcal{Q}11$	+		+		+	+		$\mathcal{Q}38$	+	+	+	+			
$\mathcal{Q}12$	+			+				$\mathcal{Q}39$	+	+	+	+			
$\mathcal{Q}13$	+						+	$\mathcal{Q}40$	+			+	+		
$\mathcal{Q}14$	+					+	+	$\mathcal{Q}41$	+	+	+	+			
$\mathcal{Q}15$	+					+		$\mathcal{Q}42$	+			+	+		
$\mathcal{Q}16$	+					+		$\mathcal{Q}43$	+	+			+		
$\mathcal{Q}17$	+						+	$\mathcal{Q}44$	+				+		
$\mathcal{Q}18$	+	+						$\mathcal{Q}45$	+		+	+		+	+
$\mathcal{Q}19$	+			+				$\mathcal{Q}46$	+	+					
$\mathcal{Q}20$	+						+	$\mathcal{Q}47$	+	+					
$\mathcal{Q}21$	+			+	+			$\mathcal{Q}48$	+		+	+	+		
$\mathcal{Q}22$	+	+		+				$\mathcal{Q}49$	+	+	+	+			
$\mathcal{Q}23$	+			+			+	$\mathcal{Q}50$	+	+		+			
$\mathcal{Q}24$	+		+	+			+	$\mathcal{Q}51$	+	+		+			
$\mathcal{Q}25$	+	+	+	+				$\mathcal{Q}52$	+		+	+	+		
$\mathcal{Q}26$	+		+	+	+			$\mathcal{Q}53$	+	+		+			
$\mathcal{Q}27$	+			+		+		$\mathcal{Q}54$	+	+		+			

Table 4.1: Classification of all queries ($\mathcal{Q}1$- $\mathcal{Q}54$).

4.4.1 Results for enumeration phase

First, we measure the enumeration capability of the RLQDAG in terms of the number of plans explored per second for each query. The goal is to explore the plan space exhaustively, by finding every possible plan given the set of rewrite rules. Figures 4.8, 4.9, 4.10 and 4.11 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 enumerates plans much faster (up to two orders of magnitude) when compared to the μ-RA system[1].

---

[1]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.
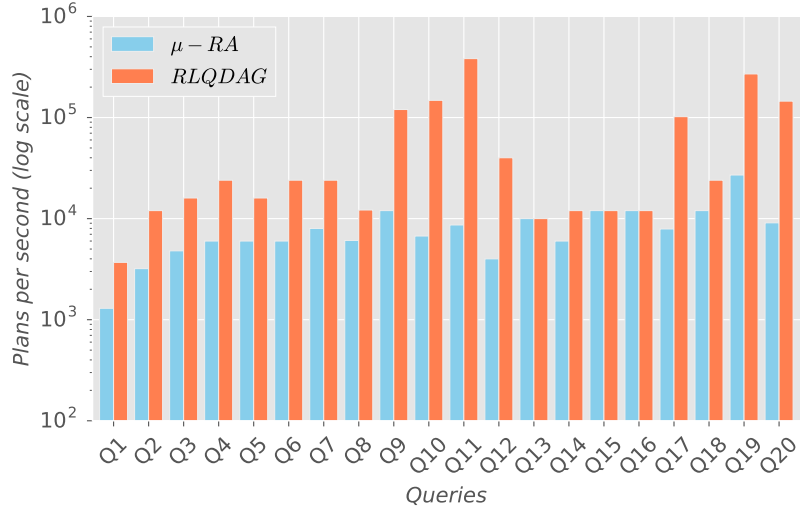
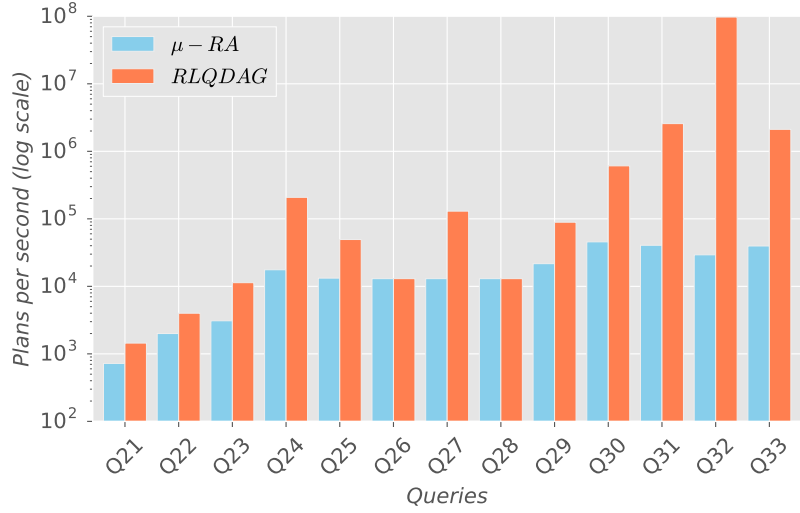Figure 4.8: Plan space explored for Yago queries.

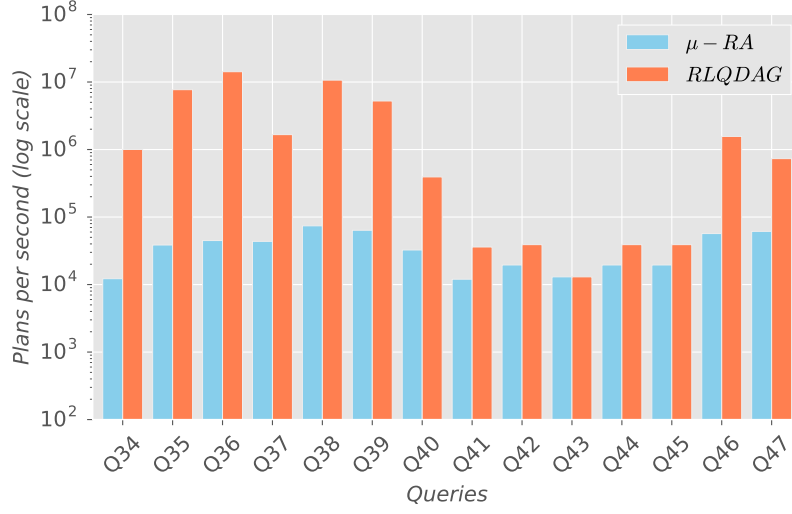Figure 4.9: Plan space explored for Bahamas Leaks queries.

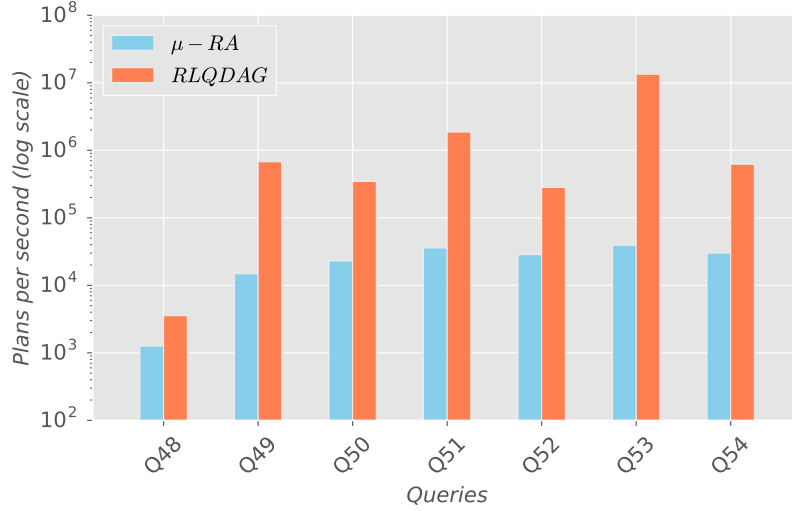Figure 4.10: Plan space explored for Airbnb queries.

Figure 4.11: Plan space explored for LDBC queries.

Plan space exploration in a time budget of 0.5 seconds.

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 portions of plan spaces obtained by the two systems. Figure 4.12 shows the results obtained for a time budget of $t = 0.5$ seconds. The y axis (in log scale) indicates the number of plans found. Figure 4.12 also indicates the size of the complete (exhaustive) plan space obtained without any time restriction for the plan exploration ($t = \infty$). For example, for query Q31, the complete plan space contains more than 21.4 million plans. In 0.5 seconds, the RLQDAG prototype explored 1,019,026 plans whereas the μ-RA system explored only 5,751 plans. This is because although μ-RA uses dynamic programming techniques, it is not capable of benefitting from the RLQDAG's grouping effect when applying complex rewrite rules on sets of

recursive terms at once, thus rules are significantly more costly to apply. Seen from another perspective, this means that the RLQDAG's approach is more effective in avoiding redundant subcomputations. We have conducted extensive experiments and overall results indicate that, for a given time budget, the RLQDAG prototype explores many more terms in comparison to the μ-RA system, 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 Q32, Q51, Q53, Q54 in Figure 4.12).
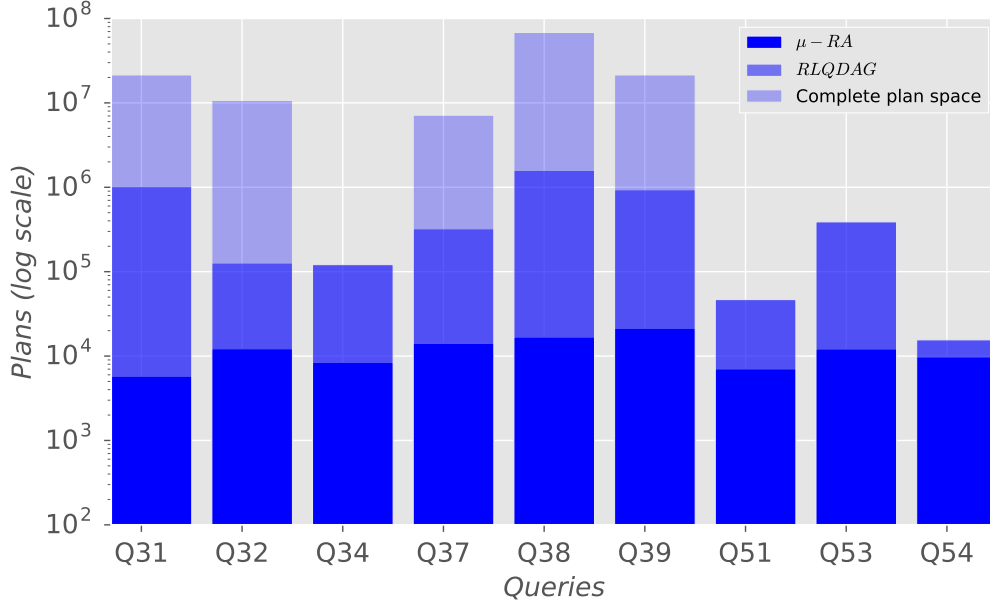
Figure 4.12: Plan spaces explored in a fixed time budget (0.5 s).

Plan space explored in a given time budget (Q31)

We study how the plan space explored evolves when given different time budgets.

In the example of Figure 4.13, we check the plan space portion explored after each increment of 100ms of the exploration time budget, starting from 100ms and up to 1s. In this example we considered Q31 because the number of plans explored is relatively big and therefore we can study the evolution of the plan space portions explored by each system. On Figure 4.13, we observe that the ratio between the number of plans explored by RLQDAG and μ-RA prototypes is up to 3 orders of magnitude ($\tilde{9}99.000$), which is significant. Even with the increase of the exploration time budget, this ratio stays stable.

Figure 4.13: Plan space generation in a given time budget.

In the next experiments we study to which extent these additional plan space portions explored are useful in practice.

4.4.2   Results for query evaluation phase

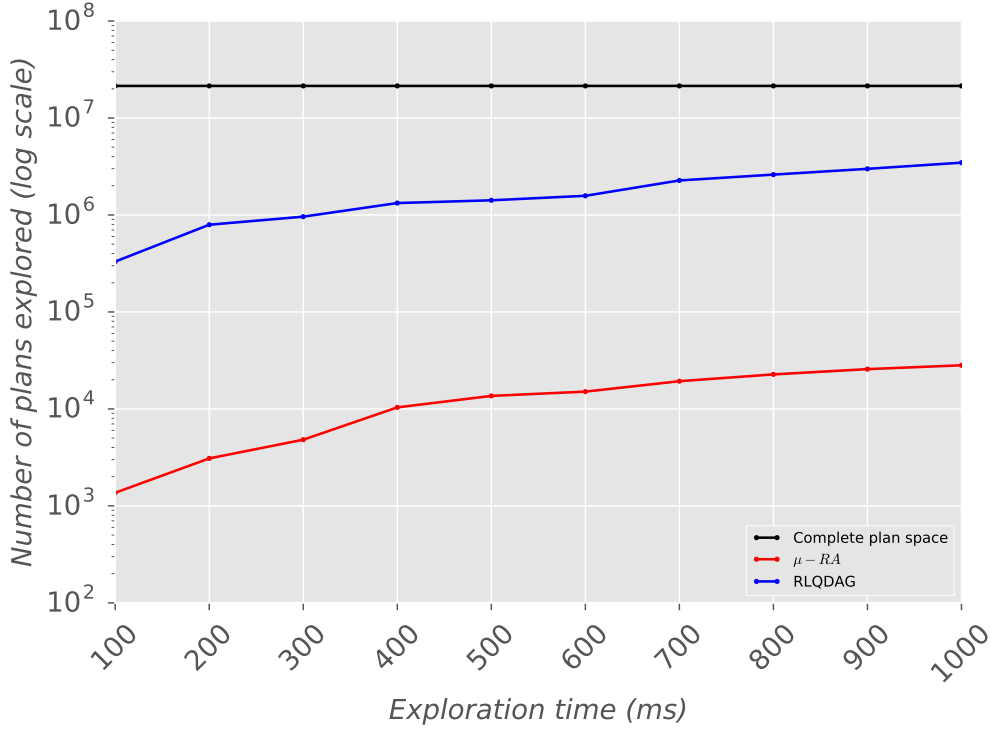We want to assess the impact of the usage of RLQDAG in the complete query evaluation process. We compare the RLQDAG and μ-RA systems while using the same heuristics for cost estimations [Lawal et al., 2020], thus making relevant a head to head comparison.

Best estimated term in a generated plan space for several queries
Figure 4.14 illustrates the time spent in evaluating the best estimated plan taken from each of the explored plan space portions of Figure 4.12. Results presented in Figure 4.14 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 explored portion of the plan space.

Best estimated term for different time budgets
We now report on experiments of exploring plan spaces with varying and increasing time budgets for the same query. To illustrate this, we consider two queries from two different datasets. For instance, Figure 4.15a presents the number of plans explored (on the y axis) for different time budgets shown on the x axis for query Q31. Figure 4.15b shows the time spent in evaluating the best estimated
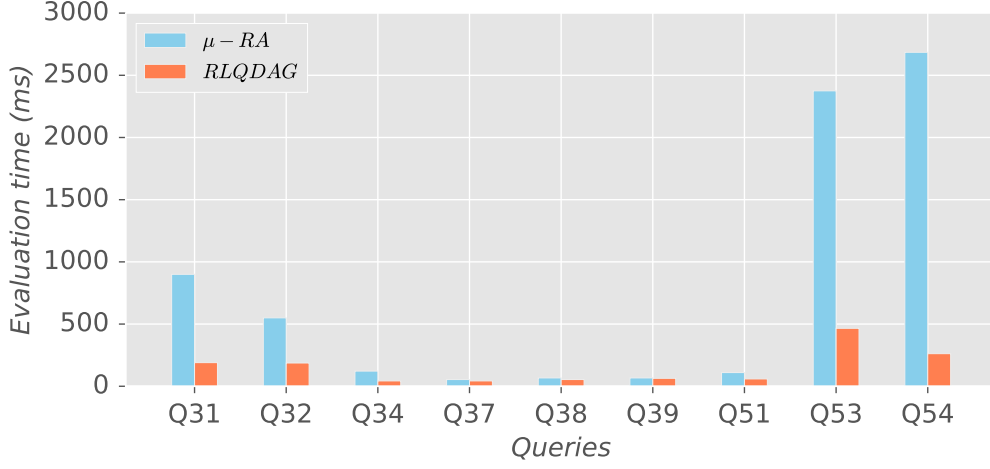
Figure 4.14: Evaluation of the most efficient plan found in the explored space for the given queries.

plan selected from each plan space explored with the time budget shown on the x axis. Again, results shown in Figure 4.15a indicate that the RLQDAG explores significantly more plans than the other system 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. Results shown in Figure 4.15b illustrate the benefits of exploring larger plan spaces since the best estimated term selected from the larger plan space is indeed more efficiently evaluated. Similar results are obtained for other queries on other datasets, as shown by e.g. in Figures 4.16a and 4.16b. This confirms the interest of efficiently exploring recursive plan spaces, since this shows that, in practice, larger plan spaces are very prone to contain more efficient recursive plans.
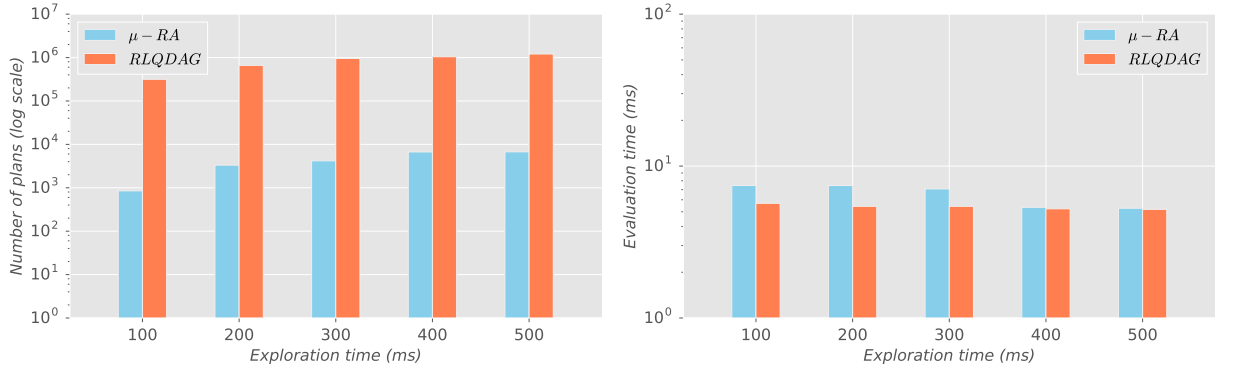
(a) Plans explored for different time budgets.   (b) Most efficient plan evaluated in the explored space.

Figure 4.15: Results on exploration and evaluation phases for Q31 in different time budgets.
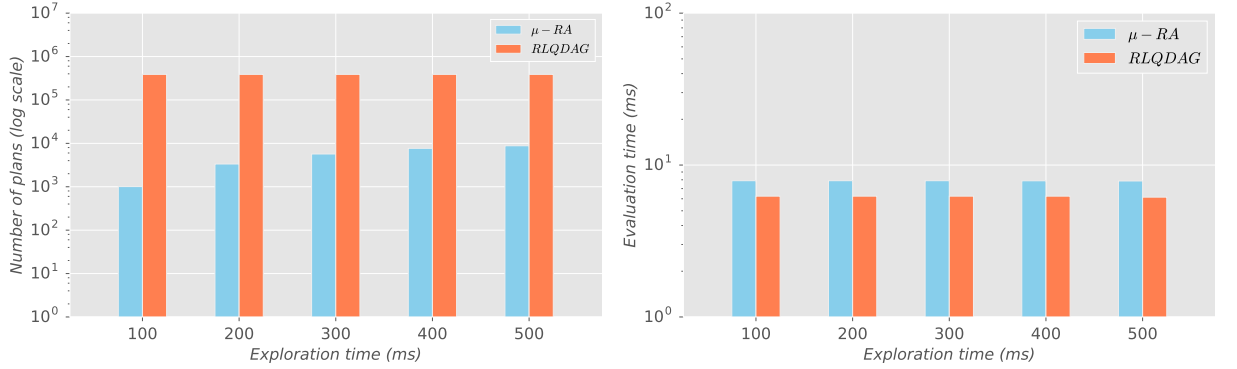
(a) Plans explored for different time budgets. (b) Most efficient plan evaluated in the explored space.

Figure 4.16: Results on exploration and evaluation phases for Q53 in different time budgets.

4.5   Conclusion

We introduced a high-level query language fragment for property graphs. We presented a syntax and translation functions for generating RLQDAG from a given query. We experiment the RLQDAG with queries on real and synthetic datasets, and compare the results obtained against a state-of-the-art baseline (μ-RA). Results show that the RLQDAG is able to enumerate plans much faster (sometimes up to 3 orders of magnitude) than the unmodified baseline system. Then, we show that the additional plans enumerated are useful because for the same time budget the RLQDAG is able to find a plan which is more efficient. In practice, this translates into performance gains when evaluating recursive queries over property graphs.

Chapter 5

Conclusion and Perspectives

5.1   Conclusion

In this thesis we have studied plan enumeration in the presence of recursion. We proposed 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 helped in detecting and fixing intricate transformational issues (due to misplaced recursive calls) in an earlier version of the RLQDAG. We believe that this formalization can also serve in other and further (R)LQDAG extensions, thus contributing to the extensibility of the top-down transformational approach.

We have introduced formal syntax and translation functions for a direct translation of a recursive query language fragment for property graphs (UCRPQ$_{\text{PG}}$) into RLQDAG. Practical experiments with a prototype implementation of the RLQDAG show that it efficiently generates the rich plan spaces introduced in the μ-recursive relational algebra. Experiments show significant performance gains compared to the state-of-the-art μ-RA system. Experiments are conducted on knowledge and property graphs. This suggests that the RLQDAG represents a promising approach for the efficient enumeration of recursive relational query plans. Based on these experiments we show how plan enumeration using RLQDAG approach can be beneficial in finding the best estimated term for a given query.

5.2   Perspectives

These works open several perspectives for future work, in different directions. We comment below on some of the main perspectives opened for further research:

5.2.1   Normal form for RLQDAG terms

Another idea is to consider a normal form of RLQDAG terms to simplify the combinatorics when enumerating recursive terms. For example, in a RLQDAG in normal form, antiprojections could be allowed to appear only at the top-level of operands of union and fixpoint operation nodes. Such a normal form could be reached by a pulling phase where antiprojections are pulled until they reach these "barriers" in the tree of operators: they would not be allowed to be pulled through these "barriers". Enumeration would then consist applying a restricted set of rewrite rules on the normal form, that thus does not need to deal with antiprojections anymore (hence decreasing the combinatorics). Finally, antiprojections would be pushed as close to the leaves (the sources) as possible. This would probably help in investigating computional complexity bounds of recursive plan enumeration, because it would be possible to have a phase of

stratification for certain operators and maybe allow an optimization considering only joins. Another idea in the same direction might be the one of disabling the strict typing system for unions and fixpoints and allow a stratification phase. This would mean we could consider an optimization phase only for join operators and we could directly compare to the propositions found in the litterature for join optimization and complexity. But disabling the strict typing system comes with huge costs that need to be considered carefully.

5.2.2 Cost estimations

An interesting perspective for further work is to investigate refinements of the cost estimation model for individual terms [Lawal et al., 2020] used to choose the most efficient plan in the setting of the RLQDAG.

5.2.3 Directed plan enumeration

As a perspective for future work it would be worth investigating the possibility to enumerate plans using the branch and bound technique. This means using a cost estimation model during the plan enumeration phase. Cost estimations would serve as a guidance for the plan enumeration phase and would allow to consider only the most efficient plans. With this, we would leverage cost estimations to prune portions of the plan space directly during the enumeration phase. This would also allow to investigate trade-offs between exhaustive plan enumeration and heuristics-based plan enumeration, in particular for fixed time budgets of optimization.

5.2.4 More expressive query language fragments

Another area of perspectives would be the consideration of more expressive fragments of high-level query languages (more expressive than UCRPQ$_{PG}$). This might include features found in different query languages available like: Cypher, PGQL, G-Core etc.

5.2.5 Benchmark on experiments

An interesting perspective is to setup a benchmark for even more advanced experiments with the RLQDAG. For now, our experiments are made against μ-RA system. This system was compared and shown to be more efficient than other systems like: neo4j, Virtuoso, LogicBlox etc. By transitivity, since RLQDAG performs better that μ-RA system we consider it as superior to these other systems. However, it would be interesting to compare it to other state-of-the-art systems. For example, there are other systems that came out very recently [Vrgoc et al., 2021] and it would be interesting to create benchmarks with them.

5.2.6 Leverage RLQDAG formalisation for theorem proving

The formal syntax and semantics proposed for RLQDAG pave the way for formal proofs for correctness and completeness using proof assistants, such as: Coq.

5.2.7   Characterization of Complexity of Expansion Algorithm

The computational complexity of plan enumeration was discussed in Section 2.3.5. Based on earlier results for SPJ queries, we can infer a lower complexity bound for plan enumeration with the RLQDAG, as illustrated in Figure 5.1.
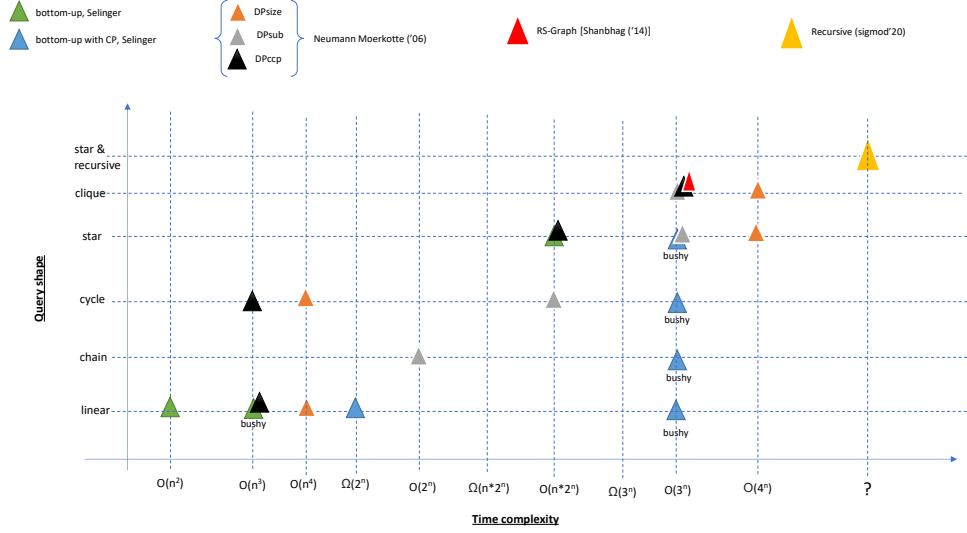
Figure 5.1: Recursive queries placement in the complexity window.

Indeed, the only thing certain for now are the lower bounds achieved when considering SPJ, but the exact complexity of plan enumeration for recursive queries is still to be studied. So far, only join relations were considered for complexity studies. If we want to study the computational complexity of recursive queries we need to take into account unions and fixpoints as well. It becomes more complicated because of rewrite rules and new possibilities generated from the interplay between operators. It would be interesting to investigate complexity upper bounds in this context.

Appendix

Queries

Queries used in experiments [Tyrex-repository, 2022] with graph datasets are shown below:

?x	←	?x isMarriedTo/livesIn/isLocatedIn+/dealsWith+ Argentina	$\mathcal{Q}_1$
?x	←	?x hasChild/livesIn/isLocatedIn+/dealsWith+ Japan	$\mathcal{Q}_2$
?x	←	?x influences/livesIn/isLocatedIn+/dealsWith+ Sweden	$\mathcal{Q}_3$
?x	←	?x livesIn/isLocatedIn+/dealsWith+ United_States	$\mathcal{Q}_4$
?x	←	?x hasSuccessor/livesIn/isLocatedIn+/dealsWith+ India	$\mathcal{Q}_5$
?x	←	?x hasPredecessor/livesIn/isLocatedIn+/dealsWith+ Germany	$\mathcal{Q}_6$
?x	←	?x hasAcademicAdvisor/livesIn/isLocatedIn+/dealsWith+ Netherlands	$\mathcal{Q}_7$
?x	←	?x isLocatedIn+/dealsWith+ United_States	$\mathcal{Q}_8$
?x	←	?x (actedIn/-actedIn)+ Kevin_Bacon	$\mathcal{Q}_9$
?area	←	wikicategory_Capitals_in_Europe -rdf:type/(isLocatedIn+/dealsWith|dealsWith) ?area	$\mathcal{Q}_{10}$
?person	←	person isMarriedTo+/owns/isLocatedIn+|owns/isLocatedIn+ United_States	$\mathcal{Q}_{11}$
?a, ?b	←	?a isLocatedIn+/dealsWith ?b	$\mathcal{Q}_{12}$
?a, ?b	←	?a isLocatedIn+/dealsWith+ ?b	$\mathcal{Q}_{13}$
?a, ?b, ?c	←	?a wasBornIn/isLocatedIn+ ?b,	
		?b isConnectedTo+ ?c	$\mathcal{Q}_{14}$
?a, ?b, ?c	←	?a (isLocatedIn|isConnectedTo)+ ?b,	
		?c wasBornIn ?a	$\mathcal{Q}_{15}$
?a, ?c	←	?a wasBornIn/IsLocatedIn+ Japan,	
		?a rdf:type/rdfs:subClassOf ?c	$\mathcal{Q}_{16}$
?a	←	?a isLocatedIn+/(isConnectedTo|dealsWith)+ Japan	$\mathcal{Q}_{17}$
?a, ?c	←	?a IsLocatedIn+ Japan,	
		?a isConnectedTo+ ?c	$\mathcal{Q}_{18}$
?a	←	?a isLocatedIn+/isLocatedIn Japan	$\mathcal{Q}_{19}$
?a	←	?a isLocatedIn+/isConnectedTo+/dealsWith+ Japan	$\mathcal{Q}_{20}$

Figure .1: Queries over the Yago dataset.

?x, ?z, ?w	←	?x: person[firstName=Carmen gender=female] person_knows_person[creationDate=2010]+/person_likes_comment[creationDate=2008] ?z: comment[browserUsed=Opera]	$\mathcal{Q}_{48}$
?x, ?y, ?z	←	?x: person[firstName=Carmen gender=female] person_knows_person[creationDate=2010]+ ?y: person[lastName=Rodriguez place=726],	
		?x: person[firstName=Carmen gender=female] person_likes_comment[creationDate=2006] ?z: comment[browserUsed=Opera]	$\mathcal{Q}_{49}$
?x, ?y, ?w	←	?x: person[lastName=Muller gender=male] person_knows_person+ ?y: person,	
		?x: person[lastName=Muller gender=male] person_likes_comment ?w: comment	$\mathcal{Q}_{50}$
?x, ?y, ?w	←	?x: person[lastName=Ivanov birthday=1986-06-08] person_likes_comment[creationDate=2012-07-31T11:25:02.077+0000] ?w: comment[browserUsed=Firefox place=57],	
		?x: person[lastName=Ivanov birthday=1986-06-08] person_knows_person+ ?y: person	$\mathcal{Q}_{51}$
?x, ?p, ?t	←	?x: person[firstName=Jun place=409] person_knows_person+/person_likes_post ?p: post[language=tk],	
		?p: post[language=tk] post_hasTag_tag ?t: tag	$\mathcal{Q}_{52}$
?x, ?y, ?w	←	?x: person[firstName=Kirill lastName=Ivanov birthday=1986-06-08] person_studyAt_organisation ?w: organisation,	
		?x: person[firstName=Kirill lastName=Ivanov birthday=1986-06-08] person_knows_person+ ?y: person	$\mathcal{Q}_{53}$
?x, ?p, ?y	←	?x: person[firstName=Jun place=409] person_likes_post ?p: post[language=tk],	
		?x: person[firstName=Jun place=409] person_knows_person+ ?y: person	$\mathcal{Q}_{54}$

Figure .2: Queries over the LDBC dataset.

```
?x, ?w          ←  ?x:  officer[name_off= ALPHA DIRECTION LTD] same_name_as+/ registered_address ?w:  address                                    Q21
?x, ?o          ←  ?x:  officer[name_off= MONTAGUE EAST LTD.] same_name_as+ ?o:  officer                                                        Q22
?x, ?o          ←  ?x:  officer[name_off= PANTILES S.A.] same_name_as+/officer_of+ ?o:  officer                                                 Q23
?x, ?o, ?w      ←  ?x:  officer[name_off= SAMSON DANIEL SIMON] same_name_as+/officer_of+ ?o:  officer,
                   ?x:  officer[name_off= PANTILES S.A.] registered_address ?w:  address[address_add=P.O. BOX N-9306, NASSAU, BAHAMAS]           Q24
?x, ?o, ?w      ←  ?x:  officer[name_off= SAMSON DANIEL SIMON] same_name_as+/officer_of+ ?o:  officer,
                   ?x:  officer[name_off= PANTILES S.A.] registered_address ?w:  address[address_add=P.O. BOX N-9306, NASSAU, BAHAMAS]           Q25
?x, ?o, ?w      ←  ?x:  officer[name_off= ALPHA DIRECTION LTD.] same_name_as+/registered_address ?w:  address[country_codes_add=BHS]             Q26
?x, ?o, ?w      ←  ?x:  officer[name_off= PANTILES S.A.] same_address_as/same_name_as+ ?o:  officer                                             Q27
?x, ?o          ←  ?x:  officer[name_off= MARTEL JULIEN D.] same_address_as[sourceid_s_add_as =Bahamas Leaks]/same_name_as+ ?o:  officer         Q28
?x, ?y, ?z      ←  ?x:  officer[name_off= WESTLAW LIMITED] officer_of ?y:  entity[ incorporation_date_ent = 09-MAR-1990],
                   ?z:  intermediary intermediary_of+ ?y:  entity[ incorporation_date_ent = 14-AUG-1992]                                        Q29
?x, ?y, ?z, ?w  ←  ?x:  officer[name_off= ZANETTI LUIGI] officer_of+ ?y:  entity[ incorporation_date_ent = 14-AUG-1992],
                   ?z:  intermediary intermediary_of ?y:  entity[ incorporation_date_ent = 08-JAN-1992],
                   ?x:  officer[name_off= ZANETTI LUIGI] registered_address ?w:  address                                                        Q30
?x, ?o, ?y, ?z  ←  ?x:  officer[name_off=ALPHA DIRECTION LTD.] same_name_as+ ?o:  officer,
                   ?x:  officer[name_off=ALPHA DIRECTION LTD.] officer_of ?y:  entity,
                   ?z:  intermediary intermediary_of ?y:  entity                                                                                Q31
?x, ?o, ?y, ?z  ←  ?x:  officer[name_off= ALPHA DIRECTION LTD.] same_name_as+ ?o:  officer,
                   ?x:  officer[name_off= ALPHA DIRECTION LTD.] officer_of ?y:  entity,
                   ?y:  entity same_company_as+ ?z:  entity                                                                                     Q32
?x, ?y, ?z, ?w  ←  ?y:  entity same_company_as+ ?z:  entity,
                   ?x:  officer[name_off= ALPHA DIRECTION LTD.] officer_of ?y:  entity,
                   ?w:  intermediary intermediary_of ?y:  entity                                                                                Q33
```

Figure .3: Queries over the Bahamas Leaks dataset.

```
?x, ?z, ?w          ←  ?x:  user[user_id=11959525 user_name=Michael] friend_of+/knows+ ?z:  host,
                       ?x:  user[user_id=11959525 user_name=Michael] wrote ?w:  review[review_id=15264060]                                       Q34
?x, ?z, ?w, ?l      ←  ?x:  user[user_id=97890 user_name=Zane] friend_of+ ?z:  user,
                       ?x:  user[user_id=97890 user_name=Zane] knows+ ?w:  host[host_name=Nathan],
                       ?w:  host owns ?l:  listing                                                                                              Q35
?x, ?z, ?w, ?l      ←  ?x:  user[user_id=97890 user_name=Zane] friend_of+ ?z:  user[user_name=Phoebe user_id=74571593],
                       ?x:  user[user_id=97890 user_name=Zane] wrote ?w:  review[review_id=32236],
                       ?w:  review[review_id=32236] reviews_for ?l:  listing[listing_id=11551]                                                  Q36
?x, ?z, ?w, ?l      ←  ?x:  user[user_id=97890 user_name=Zane] knows+ ?z:  user,
                       ?z:  user wrote ?w:  review[review_id=32236],
                       ?l:  listing[listing_id=11551] -reviews_for ?w:  review[review_id=32236]                                                 Q37
?x, ?z, ?w, ?l      ←  ?x:  host[host_id=2405795] knows+ ?z:  user[user_name=Phoebe user_id=74571593],
                       ?x:  host[host_id=2405795] friend_of+ ?w:  host,
                       ?z:  user[user_name=Phoebe user_id=74571593] wrote ?l:  review                                                           Q38
?x, ?z, ?a, ?l      ←  ?x:  host[host_id=2405795] knows+ ?z:  user[user_name=Phoebe user_id=74571593],
                       ?x:  host[host_id=2405795] owns ?l:  listing,
                       ?l:  listing has ?a:  amenity[bedrooms=3 beds=3]                                                                         Q39
?x, ?z, ?w          ←  ?x:  user[user_id=93896] friend_of+/wrote ?z:  review,
                       ?x:  user[user_id=93896] knows+ ?w:  host[host_location=England host_since=2009-12-05]                                    Q40
?x, ?z              ←  ?x:  user[user_name=Monique] friend_of+|wrote ?z:  review[comment=Super perfect 5 star place to stay]                     Q41
?x, ?y              ←  ?x:  user[user_id=100872 user_name=Sarah] friend_of+/-friend_of ?y:  user                                                Q42
?x, ?y              ←  ?x:  host friend_of+/owns ?y:  listing                                                                                   Q43
?x, ?y              ←  ?x:  host friend_of+/owns[hostID=43039 listID=11551] ?y:  listing                                                        Q44
?x, ?y              ←  ?x:  host[host_name=Adriano] owns[hostID=43039 listID=11551]/friend_of+ ?y:  listing[name=NICE FAMILY HOME opposite NATURAL LANDSCAPED PARK]   Q45
?x, ?y, ?l, ?z      ←  ?x:  user knows+ ?y:  host,
                       ?y:  host owns ?l:  listing,
                       ?l:  listing -reviews_for ?z:  review[comment = The flat was bright review_id=30672]                                     Q46
?x, ?y, ?l, ?w      ←  ?x:  host knows+ ?y:  host,
                       ?x:  host owns ?l:  listing[name=Arty],
                       ?l:  listing[name=Arty] has ?w:  booking_details[listing_id=1000127]                                                     Q47
```

Figure .4: Queries over the Airbnb dataset.

Bibliography

[dat, 2021] (2021). *PODS'21: Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, New York, NY, USA. Association for Computing Machinery.

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V. (1995). *Foundations of Databases*. Addison-Wesley.

[Abul-Basher et al., 2017a] Abul-Basher, Z., Yakovets, N., Godfrey, P., Ghajar-Khosravi, S., and Chignell, M. H. (2017a). 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*.

[Abul-Basher et al., 2017b] Abul-Basher, Z., Yakovets, N., Godfrey, P., Ghajar-Khosravi, S., and Chignell, M. H. (2017b). 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*.

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

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

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

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

[Angles et al., 2018] Angles, R., Arenas, M., Barceló, P., Boncz, P., Fletcher, G., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J., et al. (2018). G-core: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1421–1432.

[Angles et al., 2017] Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Comput. Surv.*, 50(5).

[Bahamas-Leaks, 2017] Bahamas-Leaks (2017). Bahamas leaks. https://www.kaggle.com/datasets/zusmani/paradisepanamapapers.

[Bancilhon et al., 1985] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. (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*, PODS '86, page 1–15, New York, NY, USA. Association for Computing Machinery.

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

[Barceló et al., 2013] Barceló, P., Figueira, D., and Libkin, L. (2013). Graph Logics with Rational Relations. *Logical Methods in Computer Science*, 9(3).

[Barceló et al., 2012] Barceló, P., Libkin, L., Lin, A. W., and Wood, P. T. (2012). Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4).

[Begoli et al., 2018] Begoli, E., Camacho-Rodríguez, J., Hyde, J., Mior, M. J., and Lemire, D. (2018). Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 221–230, New York, NY, USA. Association for Computing Machinery.

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

[Bonifati et al., 2018] Bonifati, A., Fletcher, G., Voigt, H., and Yakovets, N. (2018). *Querying graphs*. Morgan & Claypool Publishers.

[Casel and Schmid, 2021] Casel, K. and Schmid, M. L. (2021). Fine-grained complexity of regular path queries. In Yi, K. and Wei, Z., editors, *24th International Conference on Database Theory, ICDT 2021, March 23-26, 2021, Nicosia, Cyprus*, volume 186 of *LIPIcs*, pages 19:1–19:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

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

[Chlyah et al., 2022] Chlyah, S., Gesbert, N., Genevès, P., and Layaïda, N. (2022). On the Optimization of Iterative Programming with Distributed Data Collections. working paper or preprint.

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

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

[Cyganiak et al., 2014] Cyganiak, R., Wood, D., Lanthaler, M., Klyne, G., Carroll, J. J., and McBride, B. (2014). Rdf 1.1 concepts and abstract syntax. *W3C recommendation*, 25(02):1–22.

[DeHaan and Tompa, 2007] DeHaan, D. and Tompa, F. W. (2007). Optimal top-down join enumeration. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, page 785–796, New York, NY, USA. Association for Computing Machinery.

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

[Fender and Moerkotte, 2011] Fender, P. and Moerkotte, G. (2011). A new, highly efficient, and easy to implement top-down join enumeration algorithm. pages 864 – 875.

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

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

[Fender et al., 2012] Fender, P., Moerkotte, G., Neumann, T., and Leis, V. (2012). Effective and robust pruning for top-down join enumeration algorithms. In *2012 IEEE 28th International Conference on Data Engineering*, pages 414–425.

[Francis et al., 2018a] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., and Taylor, A. (2018a). Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference*, pages 1433–1445. ACM.

[Francis et al., 2018b] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., and Taylor,

A. (2018b). Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA. Association for Computing Machinery.

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

[Garcia-Molina et al., 2008] Garcia-Molina, H., Ullman, J. D., and Widom, J. (2008). Database systems: The complete book.

[Gardarin, 1987] Gardarin, G. (1987). Magic functions: A technique to optimize extended datalog recursive programs. pages 21–30.

[Godfrey et al., 2017] Godfrey, P., Yakovets, N., Abul-Basher, Z., and Chignell, M. H. (2017). WIREFRAME: Two-phase, cost-based optimization for conjunctive regular path queries. In *AMW*.

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

[Graefe, 1995] Graefe, G. (1995). The cascades framework for query optimization. *Data Engineering Bulletin*, 18.

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

[Gremlin, 2022] Gremlin (2022). Gremlin. https://tinkerpop.apache.org/gremlin.html.

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

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

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

[Ibaraki and Kameda, 1984] Ibaraki, T. and Kameda, T. (1984). On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502.

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

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

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

[Libkin et al., 2016] Libkin, L., Martens, W., and Vrgoč, D. (2016). Querying graphs with data. *J. ACM*, 63(2).

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

[Moerkotte and Neumann, 2006] Moerkotte, G. and Neumann, T. (2006). Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 930–941. VLDB Endowment.

[Moerkotte and Neumann, 2008] Moerkotte, G. and Neumann, T. (2008). Dynamic programming strikes back. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 539–552, New York, NY, USA. Association for Computing Machinery.

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

[Neo4j, 2007] Neo4j (2007). Neo4j. https://neo4j.com/.

[Neumann and Radke, 2018] Neumann, T. and Radke, B. (2018). Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 677–692, New York, NY, USA. Association for Computing Machinery.

[Ono and Lohman, 1990] Ono, K. and Lohman, G. M. (1990). Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, page 314–325, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Pellenkoft et al., 1997] Pellenkoft, A., Galindo-Legaria, C., and Kersten, M. (1997). The complexity of transformation-based join enumeration. pages 306–315.

[Pirahesh et al., 1992] Pirahesh, H., Hellerstein, J. M., and Hasan, W. (1992). Extensible/rule based query rewrite optimization in starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, SIGMOD '92, page 39–48, New York, NY, USA. Association for Computing Machinery.

[Reutter et al., 2017] Reutter, J. L., Romero, M., and Vardi, M. Y. (2017). Regular queries on graph databases. *Theor. Comp. Sys.*, 61(1):31–83.

[Roy, 2001] Roy, P. (2001). *Multi Query Optimization and Applications*. PhD thesis, PhD thesis, Indian Institute of Technology, Bombay.

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

[Saccà and Zaniolo, 1985] Saccà, D. and Zaniolo, C. (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*, PODS '86, page 16–23, New York, NY, USA. Association for Computing Machinery.

[Seaborne et al., 2008] Seaborne, A., Manjunath, G., Bizer, C., Breslin, J., Das, S., Davis, I., Harris, S., Idehen, K., Corby, O., Kjernsmo, K., et al. (2008). Sparql/update: A language for updating rdf graphs. *W3c member submission*, 15.

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

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

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

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

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

[Soliman et al., 2014] Soliman, M. A., Antova, L., Raghavan, V., El-Helw, A., Gu, Z., Shen, E., Caragea, G. C., Garcia-Alvarado, C., Rahman, F., Petropoulos, M., Waas, F., Narayanan, S., Krikellas, K., and Baldwin, R. (2014). Orca: A modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 337–348, New York, NY, USA. Association for Computing Machinery.

[Steinbrunn et al., 1997] Steinbrunn, M., Moerkotte, G., and Kemper, A. (1997). Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal*, 6(3):191–208.

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

[TinkerPop, 2022] TinkerPop, A. (2022). Apache tinkerpop. https://tinkerpop.apache.org/.

[Tyrex-repository, 2022] Tyrex-repository (2022). Datasets and queries used in experiments with the μ-LQDAG. https://gitlab.inria.fr/tyrex-public/rlqdag.

[van Rest et al., 2016] van Rest, O., Hong, S., Kim, J., Meng, X., and Chafi, H. (2016). Pgql: A property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, GRADES '16, New York, NY, USA. Association for Computing Machinery.

[Vance and Maier, 1996] Vance, B. and Maier, D. (1996). Rapid bushy join-order optimization with cartesian products. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 35–46, New York, NY, USA. Association for Computing Machinery.

[Vardi, 1982] Vardi, M. Y. (1982). The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146.

[Vrgoc et al., 2021] Vrgoc, D., Rojas, C., Angles, R., Arenas, M., Arroyuelo, D., Aranda, C. B., Hogan, A., Navarro, G., Riveros, C., and Romero, J. (2021). Millenniumdb: A persistent, open-source, graph database. *arXiv preprint arXiv:2111.01540*.

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

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

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

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

[Yakovets et al., 2015b] Yakovets, N., Godfrey, P., and Gryz, J. (2015b). Waveguide: Evaluating SPARQL property path queries. In *EDBT*, volume 2015, pages 525–528.