On Algebraic Foundations for the Optimization of Iterative Programming with Distributed Data Collections

Sarah Chlyah

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Sarah CHLYAH

Thèse dirigée par **Pierre GENEVES**
et codirigée par **Nabil LAYAIDA**,Chercheur, Université Grenoble Alpes

préparée au sein du **Laboratoire Institut National de Recherche en Informatique et en Automatique**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Fondements algébriques pour l'optimisation de la programmation itérative avec des collections de données distribuées

On Algebraic Foundations for the Optimization of Iterative Programming with Distributed Data Collections

Thèse soutenue publiquement le **23 mai 2022**,
devant le jury composé de :

Monsieur Pierre GENEVES
DIRECTEUR DE RECHERCHE, CNRS, Directeur de thèse
Monsieur Nabil Layaida
DIRECTEUR DE RECHERCHE, Inria, Co-directeur de thèse
Monsieur Dario Colazzo
PROFESSEUR, Université Paris-Dauphine, Rapporteur
Madame Angela Bonifati
PROFESSEUR, Université Lyon, Rapporteure
Monsieur Noel De Palma
PROFESSEUR, Université Grenoble Alpes, Président du jury et Examinateur
Madame Sara Bouchenak
PROFESSEUR, INSA Lyon, Examinatrice

Abstract

The goal of my PhD is to study the optimization and the distribution of queries, especially recursive queries, handling large amounts of data. I start by reviewing different query languages as well as formal approaches to intermediate representations of these languages. Languages and formal approaches are reviewed in the light of a number of aspects such as expressivity, distribution, automatic optimizations, manipulating complex data, graph querying, and impedance mismatch, with a special focus on the ability to express recursion.

I then propose extensions to formal approaches along two main lines of work: (1) algebras based on the relational model, for which I propose Dist-μ-RA, and (2) algebras based on generic collections of arbitrary types, for which I propose μ-monoids.

Dist-μ-RA is a system that extends the μ-RA algebra to the distributed setting. Regarding the algebraic aspect, it integrates well with the relational algebra and inherits its advantages including the fact that queries are optimized regardless of their initial shape and translation into the algebra. With respect to distribution, different strategies for evaluating recursive algebraic terms in a distributed setting have been studied. These strategies are implemented as plans with automated techniques for distributing data in order to reduce communication costs. Experimental results on both real and synthetic graphs show the effectiveness of the proposed approach compared to existing systems.

μ-monoids is an extension of the monoid algera with a fixpoint operator that models recursion. The extended μ-monoids algebra is suitable for modeling recursive computations with distributed data collections such as the ones found in Big Data frameworks. The major interest of the "μ" fixpoint operator is that, under prerequisites that are often met in practice, it can be considered as a monoid homomorphism and thus can be evaluated by parallel loops with one final merge rather than by a global loop requiring network overhead after each iteration. Rewriting rules for optimizing fixpoint terms, such as pushing filters, are proposed. In particular, I propose a sufficient condition on the repeatedly evaluated term (φ) regardless of its shape, as well as a method using polymorphic types and a type system such as Scala's to check whether this condition holds. I also propose a rule to prefilter a fixpoint before a join. The third rule allows for pushing aggregation functions inside a fixpoint. Experiments with the Spark platform illustrate performance gains brought by these systematic optimizations.

Résumé

Le but de ma thèse est d'étudier l'optimisation et la distribution de requêtes, principalement de requêtes récursives, qui manipulent de larges volumes de données. Premièrement, je passe en revue différents langages de requêtes ainsi que différentes approches formelles liées aux représentations intermédiaires de ces langages. Les langages et les approches formelles sont examinés à la lumière d'un nombre d'aspects tels que l'expressivité, la distribution, les optimisations automatiques, la manipulation des données complexes, le requêtage de graphes, l'incompatibilité d'impédance, avec une attention particulière portée à la capacité à exprimer des requêtes récursives.

Dans un second temps, je propose des extensions d'approches formelles suivant deux axes de travaux de recherche: (1) les algèbres basées sur le modèle de données relationnel et pour lesquelles je propose Dist-μ-RA, ainsi que (2) les algèbres basées sur les collections de types arbitraires, et pour lesquelles je propose μ-monoids.

Dist-μ-RA est un système qui étend l'algèbre μ-RA au contexte distribué. Concernant l'aspect algébrique, il s'intègre bien avec l'algèbre relationnelle et hérite de ses avantages tels que sa capacité à optimiser les requêtes quelles que soient leur forme initiale et leur traduction vers l'algèbre. Concernant l'aspect de distribution, différentes stratégies d'évaluation de termes algébriques récursifs dans un contexte distribué ont été étudiées. Ces stratégies sont implémentées sous forme de plans physiques avec des techniques qui automatisent la distribution des données afin de réduire les coûts de communication. Les résultats expérimentaux sur des graphes réels et synthétiques montrent l'efficacité de l'approche proposée par rapport aux systèmes existants.

μ-monoids est une extension de l'algèbre de monoïdes avec un opérateur de point fixe qui modélise la récursion. L'algèbre μ-monoids est capable de modéliser des calculs récursifs sur des collections distribuées similaires à ceux effectués sur les plateformes Big Data. L'intérêt principal de l'opérateur de point fixe "μ" est que, sous réserve de conditions souvent remplies en pratique, il peut-être considéré comme un homomorphisme de monoïdes et peut donc être évalué avec des boucles parallèles avec une fusion finale plutôt qu'avec une boucle globale nécessitant des transferts réseau supplémentaires à chaque itération. Des règles de réécriture pour optimiser les termes récursifs, telles que le poussage de filtres, ont été proposées. Je propose en particulier une condition suffisante sur le terme évalué en boucle (φ) quelle que soit sa forme, ainsi qu'une méthode qui utilise les types polymorphes et un système de types comme celui de Scala pour vérifier si cette condition est remplie. Je propose également une règle qui préfiltre les points fixes avant les jointures. La troisième règle permet de pousser des fonctions d'agrégation dans les points fixes. Les expériences menées sur la plateforme Spark montrent les gains en performances apportés par ces optimisations systématiques.

Remerciements

Je tiens à remercier tous ceux qui on participé de près ou de loin à l'acheminement de cette thèse.

Je voudrais d'abord remercier Pierre Genevès et Nabil Layaïda pour m'avoir aidée et encadrée tout au long de ma thèse. Merci pour votre encouragement, vos conseils et votre confiance en moi. C'était une chance d'avoir travaillé avec vous. Je voudrais aussi remercier Nils Gesbert pour son aide et l'éclairage qu'il m'a apporté sur ces parties de ma thèse qui font mal au cerveau, mais également pour toutes les discussions très enrichissantes sur les mathématiques fondamentales et l'informatique théorique, sans oublier l'excellent Baba Is You :)

Je tiens à remercier Angela Bonifati et Dario Colazzo pour avoir accepté de lire ce manuscrit ainsi que Sara Bouchenak et Noel De Palma pour avoir accepté d'examiner ma thèse. Je remercie également tous les membres du Jury pour avoir répondu présent à ma soutenance.

Merci Amela pour m'avoir accompagnée dans cette aventure aussi misérable que passionnante qu'est le doctorat.

Je remercie chaleureusement tous les membres de l'équipe Tyrex que j'ai rencontrés depuis le début de ma thèse pour l'atmosphère accueillante et conviviale et pour toutes les pauses café. Ces petits moments de répit, à peine le temps de boire une tasse de café et de discuter de météo, de géopolitique, de morale, de cuisine et de cinéma.

Merci à ma nouvelle équipe SED pour votre accueil, votre sympathie, ainsi que ces petits moments d'échange malgré les contraintes de la pandémie.

Enfin, à mes proches qui m'ont soutenue tout le long, grâce votre support sans faille et votre foi j'ai pu tenir bon. Quoique je dise je ne vous remercierai jamais assez.

Introduction

With the proliferation of large scale datasets of various data structures (such as graphs, collections, documents, trees, etc.) and in various domains (such as knowledge representation, social networks, transportation, biology, etc.), the need for efficiently extracting information from these datasets becomes increasingly important. This requires the development of methods for effectively distributing both data and computations so as to enable scalability and improve perfomance. Efforts to address these challenges over the past few years have led to various systems such as MapReduce [Dean and Ghemawat, 2004], Dryad [Isard et al., 2007], Spark [Zaharia et al., 2016], Flink [Carbone et al., 2015] and more specialized graph systems like Google Pregel [Malewicz et al., 2010], Giraph [gir, 2019] and Spark Graphx [Gonzalez et al., 2014]. While these systems can handle large amounts of data and allow users to write a broad range of applications, they still require significant programmer expertise. The system programming paradigms and its underlying configuration tuning must be highly mastered. This includes for example figuring out how to (re)partition data on the cluster, when to broadcast data, in which order to apply operations in order to reduce data transfers between nodes of the cluster, etc. So, building data extensive applications remains a very timeconsuming and expensive task.

One approach to this problem is to offer the user a Domain Specific Language (DSL) to query the data. A DSL is a high level language that is specialized in a particular application domain, and that can be called from within a general purpose language. Queries in this DSL would be translated to an intermediate representation (e.g. an algebra) so that they can be optimized automatically. The idea is to relieve users from having to worry about optimization in the distributed setting, so that they can focus only on formulating domain-specific queries in a declarative manner. A notoriously successful example of this approach is the SQL language and its associated Relational Algebra. This success is due to the level of abstraction provided by the declarative syntax of SQL as well as the extensively studied optimizations provided by Relational Algebra. In RA, data is modelled as relations made of rows and columns. This means that more complex user defined data (data defined by the user in the general purpose programming language) has to be flattened to match the tabular data model of the DSL. This illustrates what is known as the *impedance mismatch* problem. In addition, custom transformations on data can either be done with extensions like PL/SQL or in the programming language which exacerbates the impedance mismatch problem and prevents holistic program optimization.

It is then important to investigate intermediate representations for expressing and optimizing queries that manipulate data in their native format. As argued by Meijer in [Meijer and Bierman, 2011], establishing and standardizing a formal background for the noSQL market, which now contains multiple separate systems and solutions, is necessary for its economic growth as it was the case for the SQL market thanks to the introduction of RA. The author considers that an algebra based on *monads* is a suitable formalism for this purpose. Studying intermediate representations that allow for expressing operations on data in their native format would also pave the way for optimizing subsets of general purpose languages and embedded DSLs that do not suffer from impedance mismatch problems.

Motivated by the abundance of interconnected data (Web data, RDF graphs, networks of various kinds, ...) and the recent hype around knowledge graphs [Arenas et al., 2021], recursion is an active research topic [Reutter et al., 2017, Libkin and Vrgoč, 2012, Jachiet et al., 2020]. A significant class of big data programs are iterative or recursive in nature (PageRank, k-means, shortest-path, reachability, etc.). Recursion is also a very important feature for graph querying as it enables to navigate through the graph and express traversal queries such as paths of arbitrary length. More generally, it enables to model cyclical relations between entities, recursively derive new knowledge from a given initial knowledge, and extract useful information based on connectivity from the graph. Recursive queries on large-scale graphs can be very costly or even infeasible. This is due to a large combinatorial of basic computations induced by both the query and the graph topology. Recursive queries can generate intermediate results that are orders of magnitude larger than the size of the initial graph. For example, a query on a graph of millions of nodes can generate billions of intermediate results. Therefore, being able to optimize queries and reduce the size of intermediate results as much as possible becomes crucial.

Contributions In this manuscript, we investigate the problem of querying large datasets, especially graphs, with a focus on recursion by studying two algebras: (1) μ-RA which is an extension of RA with a fixpoint operator. It offers a good balance between expressivity and possible optimizations of recursion, in addition to the optimizations available in RA. (2) *monoid algebra* in which data is modelled as distributed homogeneous collections of arbitrary types. It defines operations on these collections that are monoid homomorphisms, which means that they can be broken down to the application of an associative operator. This associativity implies that parts of the computation can actually be performed in parallel and combined to get the final result.

μ-RA is limited to the centralized setting, so we propose Dist-μ-RA which builds on μ-RA and extends it with optimized evaluation plans suited for the distributed setting. Experimental results on both real and synthetic graphs show the effectiveness of the proposed approach compared to existing systems. As for monoid algebra, no optimization for recursive computations is available. We propose μ-monoids, which extends monoid algebra with a fixpoint operator for representing recursion as a first class citizen and show how it enables new opti-

mizations. Experiments illustrate performance gains brought by these systematic optimizations.

Outline This dissertation is structured into two parts: State of the art (Part I) and Contribution (Part II). We start by reviewing DSLs that are used to query data in chapter 1. In chapter 2, we investigate different algebras and the optimizations they provide. We then present Dist-μ-RA in chapter 3 and μ-monoids in chapter 4. We end with a conclusion and perspectives for future works.

Contents

Part I

State of the art

Chapter 1

Domain specific languages for querying graphs and distributed data

A Domain Specific Language (DSL) is a programming language that is designed to target a specific application domain. A DSL can be distinguished from a General Purpose Language (GPL) in that it offers a specialized (hence usually limited) and preferably a convenient syntax for representing and reasoning about its domain. The goal is to allow for expressing domain problems in a simple and concise manner while abstracting away frow how the solution is computed or implemented.

DSLs can be divided into two categories: External (or standalone) and internal (or embedded) DSLs. An external DSL defines its own syntax and semantics and hence it is independent from any programming language it might be called from. SQL is an example of an external DSL. An internal DSL on the other hand relies on a general purpose language (called *the host language*) in which it is embedded. It can reuse the host language syntax and infrastructure (such as the compiler and type checker) to be built. In general, the syntax of internal DSLs is a subset of its host language syntax. However, there are languages which define their syntax as a mixture of host language expressions and new constructs that are not valid in the host language. They use metaprogramming techniques (like quotations and reflection) to allow the definition of the new syntax in the host language and to manipulate host language terms. Such languages can still be considered internal since they require to be embedded in a host language and use its infrastructure to define their language.

While external DSLs present the advantage of being independant from the language used for development, they are harder to build than internal DSLs since they require the implementation of an entire language from scratch starting from the parser down to the execution of terms. This independence also often means that the data model used in the host language is different from the one used in the DSL. This implies that data defined by the user in the programming language needs to be transformed to fit in the data model of the DSL. This is known as

the *impedance mismatch* problem. A deeper integration with the host language is possible for internal DSLs because of the possiblity they have to reuse the host language data types as well as its syntax for processing data.

There is an abundance of DSLs in the literature. Moreover, it is becoming common for software developers to build their own DSLs as part of good programming practice. In the next section we will only be focusing on DSLs that are close from our domain of interest, that is big data querying. For this, we review languages that are used for querying relational databases, graphs, and large distributed data. The term query language is usually used to refer to such languages.

1.1 Relational database querying

SQL is the most prominent language for database querying. It is an external DSL that uses the standard `SELECT FROM WHERE` syntax for querying data. The example of Fig. 1.1 is an SQL query that extracts the names, ages, and city of people that are friends and that live in the same city.

```
SELECT p1.name, p1.age, p2.name, p2.age, a1.city
FROM person AS p1 JOIN address AS a1 JOIN friends JOIN person AS p2 JOIN
    address AS a2
WHERE p1.id == friend.id1 AND p2.id == friend.id2 AND p1.address_id ==
    a1.id AND p2.address_id == a2.id AND a1.city == a2.city
```

Figure 1.1: SQL query example.

The success of SQL is due to the level of abstraction provided by its declarative syntax as well as the optimizations provided by Relational Algebra on which it is based. However, it suffers from the impedance mismatch problem mentioned earlier since user defined data have to be flattened to match the tabular data model of SQL. In the example above, the information about people and their friendship relationships is flattened to three tables: person, friends and address. In addition, it provides limited support for complex data processing (data transformation, iteration, aggregation, etc.). In order to perform custom transformations on data for instance, one could use language extensions like PL/SQL which, in addition to exacerbating the impedance mismatch problem, requires user expertise and provides only limited optimizations. Alternatively, the user could perform data transformations on the query results in the programming language, which increases roundtrips between the program and the database and does not allow for holistic program optimization.

LINQ [Meijer et al., 2006] is a widely known internal DSL for querying data. The LINQ query of Fig. 1.2, as the SQL example above, extracts information about friends living in the same city. LINQ provides common querying operations like filtering, ordering, joining, as well as data transformation. However it does not support iteration. Data that can be queried include SQL databases (LINQ to SQL), XML documents or in-memory collections. Queries can be performed on

any host language collection as long as it implements the IEnumerable interface. This adresses the impedance mismatch issue posed by SQL. However, when mapped to SQL, LINQ restricts the set of host language expressions that can be used for transforming data.

```
friends
    .Where(f => f.p1.address.city == f.p2.address.city)
    .Select(f=> new {f.p1.name, f.p1.age, f.p2.name, f.p2.age, f.p1.city})
```

Figure 1.2: LINQ query example.

DSH [Giorgidze et al., 2011] is an internal DSL embedded in Haskell for querying RDBMSs. Queries are expressed as Haskell list comprehensions. A DSH query get translated into SQL queries that get executed on the RDBMS. It implements the avalanche safety property for SQL generation which guarantees that the number of generated SQL queries does not depend on size of queried data but only on the query. This was later introduced in LINQ.

1.2 Graph querying

Graphs have become one of the most prominent ways of representing data. They are used in various domains such as social networks, planning, transportation, knowledge representation, biology, machine learning, etc. [Bonifati et al., 2018, Sakr et al., 2021]. These graphs can be split to different categories depending on their data model. Each data model comes with a set of languages used to query the graphs. We first discuss the graph data models, then we present the query languages.

1.2.1 Graph data models

A graph data model is a data model where the objects of a domain are represented by nodes and relationships between them are represented by edges. A graph data model can further be specialized into different data models depending the data structures used to define nodes and edges. Labelled graphs are the most basic graph model where nodes and edges are simply assigned a label. More complex graph models are: property graphs where nodes and edges can additionally be assigned attributes, hypergraphs where edges can relate any number of nodes and the hyperedge model where nodes can themselves be graphs [Angles and Gutierrez, 2018]. We will next be focusing on the *edge-labelled graphs* and *property graphs* data models as they are the most commonly found in the literature [Angles et al., 2017].

Edge-labelled graph An edge-labelled graph is formalized as a pair (V, E) where V is a finite set of vertices and $E \subseteq V \times Lab \times V$ a finite set of labelled edges with labels belonging to a finite set Lab.

Fig. 1.3 shows an example of an edge-labelled graph.

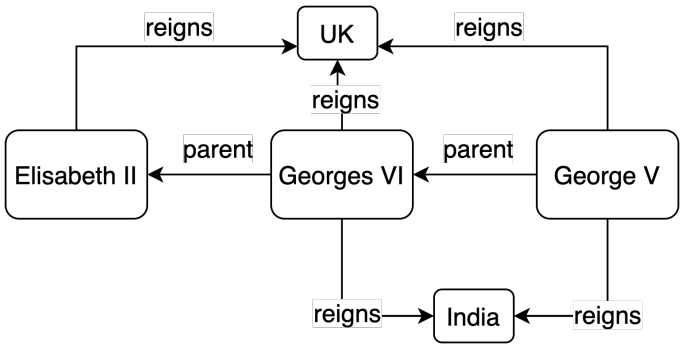


Figure 1.3: Edge-labelled graph example.

RDF graphs are a special case of such graphs. RDF is a standard for representing information in the Web [Graham Klyne, 2014].

Property graphs In property graphs, it is possible to assign attributes (property value pairs) as well as a label to both nodes and edges. Formally, it is defined as a tuple $(V, E, \rho, \lambda, \sigma)$ where :

- V a finite set of vertices and E a finite set of edges such that $V \cap E = \emptyset$
- $\rho : E \to (V \times V)$ a function that assigns to each edge its source and target nodes.
- $\lambda : V \cup E \to Lab$ a function that assigns to each vertex and edge a label in Lab, where Lab is a finite set of labels.
- $\sigma : (V \cup E) \times Prop \to Val$ is a partial function that assigns a value in Val to an entity (node or edge) for a property in $Prop$, where $Prop$ is a finite set of properties and V a finite set of values.

Fig. 1.4 shows an example of a property graph.

1.2.2 Graph query languages

Various languages are proposed to query graphs of different data models. For querying edge-labelled graphs, there is the standard query language SPARQL for RDF graphs, as well as other languages such as GraphLog and Gram. For querying property graphs, there are Cypher and Gremlin which are among the most popular languages, G-CORE and PGQL. We present below some of these languages.

SPARQL [Harris and Seaborne, 2013] is one of the earliest languages for querying RDF graphs. The query example of Fig. 1.5 asks for the ancestors of Elisabeth II and the countries they reigned over. SPARQL has a declarative syntax similar to SQL. The `SELECT` clause specifies the variables to be retrieved

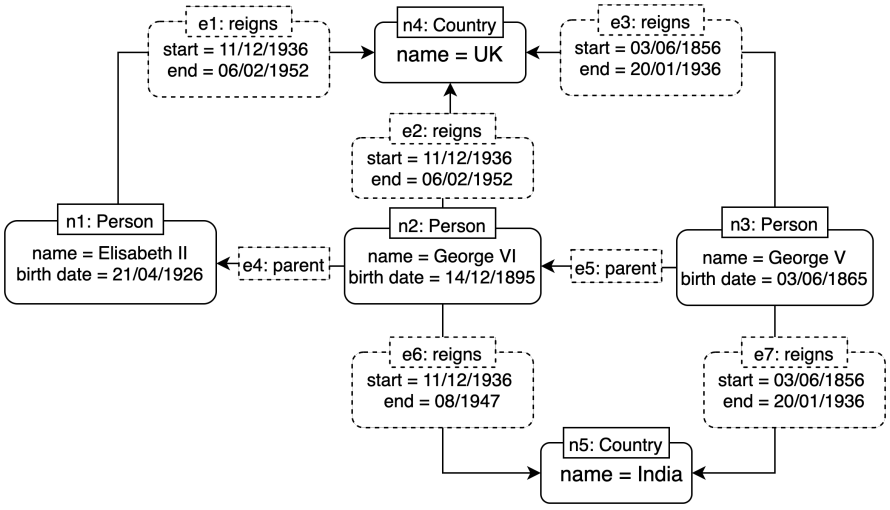


Figure 1.4: Property graph example.

```
SELECT ?x, ?y
WHERE {
    ?x parent+ Elisabeth II . ?x reigns ?y
}
```

Figure 1.5: SPARQL example.

for the query results. Variables are prefixed with ? (in the example we want to retrieve the values of the variables ?x and ?y). The `WHERE` clause is used to specify the *graph pattern* to be extracted from the queried graph. A graph pattern is a combination of *basic graph patterns* (BGP). A basic graph pattern is a set of *triple patterns*. In the example, `?x reigns ?y` is a triple pattern where `?x` is the *subject*, `reigns` is the *predicate*, and `?y` is the *object*. Subjects, objects, and predicates can be constants or variables. A BGP can be seen as a graph where nodes and edges can be variables or constants. The solutions of a BGP are obtained by matching the queried graph against this BGP (by finding the appropriate variable substitutions for which the BGP is a subset of the queried graph). This process is called *graph pattern matching*. BGPs can be combined with operations like UNION, FILTER, and OPTIONAL. In SPARQL 1.1, the querying capabilities of the language are enhanced by allowing *property paths* to be used along with triple patterns. the triple `?x parent+ Elisabeth II` is a property path. The predicate of a property path is a regular expressions on graph edges, which allows for specifying graph traversal constraints between nodes where paths can be of arbitrary length.

GraphLog [Consens and Mendelzon, 1990] is a graphical query language based on graph pattern matching. It extends the G [Cruz et al., 1987] language which first introduced the notion of a graphical query as

graphs [Angles and Gutierrez, 2018]. Fig 1.6 shows a query in Graphlog. The query is a graph (or a set of graphs to express union). The thick edge is called the

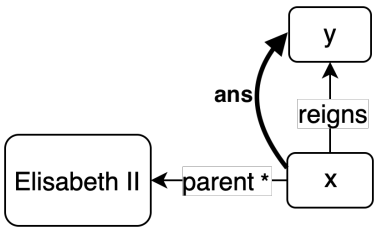


Figure 1.6: GrahLog example.

distinguished edge. It specifies the edges that are answers of the query. The query represents a graph pattern similar to that of SPARQL. Nodes can be constants or variables, and edges are regular expressions containting variables and constants.

Cypher [Francis et al., 2018] is a declarative language for property graphs. Fig 1.7 shows a query in Cypher.

```
MATCH (x:Person) -[:parent*]-> (z:Person {name:"Elisabeth II"})
MATCH (x) -[:reigns]-> (y:Country)
RETURN x.name, y.name
```

Figure 1.7: Cypher example.

The RETURN clause specifies the variable to be returned by the query. The '.' operator is used to access attributes of complex graph nodes and edges as in `x.name`. The MATCH clause specifies a basic graph pattern to be matched against. In a BGP, nodes can be filtered by label and by specifying values for some of their attributes. For instance, `z:Person {name:"Elisabeth II"}` refers to any node of label `Person` and having an attribute `name` with the value `Elisabeth II`. Like SPARQL, basic graph patterns can be unioned (UNION keyword), filtered (WHERE keyword), joined (successive MATCH clauses), and set to optional (OPTIONAL keyword). Unlike SPARQL, predicates in a MATCH clause cannot be any regular expression of edges. Arbitrary length paths are only allowed on a single edge such as `parent*` [Angles et al., 2017]. Fixed length paths over an edge are also possible, for example `?x -[:parent*2-4]-> ?y` specifies paths of length between 2 and 4 from `?x` to `?y`.

Gremlin [gre, 2022] Gremlin, unlike the other languages, has a functional style syntax. It is an embedded DSL available in languages like Java, Scala, and Python. It offers primitives that allow for traversing the graph and filtering intermediate nodes along the way. For instance, the query

```
G.V().hasLabel('Person').has('name','Alice').out('friend').in('parent')
```

gets the parents of the friends of Alice. It starts from `G.V()`, the nodes of a graph `G`. The first two calls on `G.V()` retrieve the people who are named `Alice`, the `out` primitive retrieves their friends (destinations of edges labelled `friend`) then the `in` primitive retrieves the parents of these friends (sources of edges labelled parent). Gremlin has a `match` primitive to support graph patterns that are not expressible in the above graph traversal fashion. Fig. 1.8 shows our running example query in Gremlin. It has to use the `match` primitive because the query

```
G.V().match(
    __.hasLabel('Person').has('name','Elisabeth II')
        .repeat(in('parent').hasLabel('Person')).emit().as('x')},
    __.as('x').out('reigns').hasLabel('Country')}.as('y')
)
```

Figure 1.8: Gremlin example.

needs to return the values of two variables. `match` joins the graph traversals in its arguments and returns the values for the variables declared with `as`. `repeat` allows repetition of graph traversals an unbounded number of times. For a fixed number of iterations, the `times` primitive is used following the `repeat` call.

1.2.3 Well studied query language fragments

Graph query languages, while having different syntax and functionalities, share some common features consisting of some form of graph pattern matching and graph traversal. Language fragments introduced in [Consens and Mendelzon, 1990, Cruz et al., 1987] and well studied in the literature [Reutter et al., 2017, Barceló et al., 2012, Barceló et al., 2012, Libkin et al., 2016, Bienvenu et al., 2014] capture these features with varying degrees of expressivity.

The most basic language fragment is conjunctive queries CQ. A CQ has the following syntax:

cq	::=	$(?x_1, ?x_2, ...) \leftarrow n_1\ e_1\ n'_1,\ n_2\ e_2\ n'_2,\ ...$	CQ query
n	::=	?x | c	node variable or constant

Variables are denoted by prefixing them with '?'. The right side of a CQ query (after the arrow) represents a conjunction of triples similar to a BGP in SPARQL with the difference that predicates are only constant edge labels e_i. The left side of the query specifies the variables to be returned by the query (they must all appear in the right side). For example, `(x,y)` ← `?x parent Elisabeth II, ?x reigns ?y` retrieves the parents of `Elisabeth II` and the countries they reign over. CQ lacks the ability to express paths with an unbounded number of steps which is necessary to express a query that retrieves all the ancestors of `Elisabeth II` for instance.

RPQ is a language fragment that allows for expressing the relationship between two nodes in the form of a regular expression over the graph edges. It

has the following syntax:

rpq	::=	*n p n'*	RPQ
p	::=		regular path
		e	edge label
	|	*p*−	reverse path
	|	*p*/*p*	path concatenation
	|	*p*|*p*	path union
	|	*p*+	transitive closure

For example, `?x a/b+ ?y` describes the paths from ?x to ?y that start with an edge *a* followed by one or more successive *b* edges. Combining CQ and RPQ lead to CRPQ:

$$crpq \quad ::= \quad (?x_1, ..., ?x_n) \leftarrow n_1\ p_1\ n'_1, ..., p_m \quad \text{CRPQ}$$

The CRPQ `?x, ?y ← ?x parent+ Elisabeth II, ?x reigns ?y` expresses the query of the running example.

UCRPQ is CRPQ extended with the union operator. A query in UCRPQ extracts the nodes in a graph that that verify at least one of several conjunctions of path predicates:

$$ucrpq \quad ::= \quad (?x_1, ..., ?x_n) \leftarrow cp_1 \mid ... \mid cp_m \quad \text{UCRPQ}$$

where cp_i denotes a conjunction of path predicates.

The recursion feature provided by UCRPQ is limited to transitive closure. Therefore, it cannot express other types of recursion like non-regular paths (eg. $a^n b^n$) or recursions that only compute the shortest paths.

1.3 Querying large distributed data

To handle large datasets, a number of frameworks that distribute data and computations have been created. We start by reviewing these frameworks in Sec. 1.3.1 before presenting high level query languages that target them in Sec. 1.3.2.

1.3.1 Distributed frameworks

Distributed computation frameworks became prevalent solutions for the development of large-scale data intensive applications. Over the past few years, they have been replacing relational databases for the development of applications handling large amounts of data and for which performance and scalability are critical. Examples of such systems are Hadoop MapReduce, Dryad, Spark and Flink. There are also other distributed frameworks that are designed for large-scale graph processing.

Hadoop MapReduce Hadoop MapReduce is one of earliest and widely adopted big data frameworks. In the MapReduce model, the user provides both a map function that produces key-value pairs and a reduce function that

aggregates values associated to the same key. Given these two functions, the framework handles the computation in the following way: first, input data is partitioned among a number of workers (mappers) that execute the map function on their own partition in parallel, obtained results are then stored to disk and shuffled across a number of workers (reducers) in such a way that values with the same key are on the same worker which then applies the reduce function and stores the output on HDFS. While this model is simple and fault tolerant, repetitive disk storage makes it inefficient for iterative applications.

Systems like Twister [Ekanayake et al., 2010] and Haloop [Bu et al., 2010] were later proposed to address the shortcomings of MapReduce for iterative computations by reducing the access to disk and reusing in-memory data across iterations. Later, Spark [Zaharia et al., 2016] and Flink [Carbone et al., 2015] were introduced to improve upon these systems and became prevalent for large scale and data-parallel computations.

Spark The Spark system was proposed to efficiently support data intensive applications including iterative applications while achieving scalability and fault tolerance. To give a synthesized overview of the Spark framework, let us consider the Spark program of figure 1.9 and the illustration of figure 1.10. The *driver*

```
1   X.map({case (a,b) => (a+b,a*b)}).reduceByKey(_+_).count()
```

Figure 1.9: Spark Program

is the process that runs this program. It creates tasks from calls to the Spark API (like `map` and `reduceByKey`) and sends them to be executed in parallel by *worker* nodes in the cluster. X is an *RDD* (Resilient Distributed Dataset). RDD is the main abstraction provided by Spark for distributed collections. An RDD is split to *partitions* across the workers. A partition is the basic unit of parallelism. Parallel *tasks* are run by each worker on the partition located on this worker. To be more precise, a worker can have more than one partition and a task is a unit of computation executed on the worker on a single partition. For simplicity, we consider one partition per worker in our example. The `map` operation in the example is executed in parallel as a task on each worker, which gives X_1 (Fig. 1.10). In order to execute the `reduceByKey` operation, *shuffling* (redistribution of data in Spark) is performed so that data with the same key is on the same partition. X_2 represents the result of the shuffle and X_3 is the result of `reduceByKey`. The `map` and `reduceByKey` operations are called *transformations*, which are operations that take an RDD as input and return an RDD as output. `map` is a *narrow transformation* because it does not need data from other partitions to get executed (it does not require shuffling), while `reduceByKey` is a *wide transformation*. The `count` operation is called an *action* because it returns a result (3 in the example) to the driver program. Transformations in Spark are *lazy*, which means that they do not trigger an execution. An execution is only triggered when an action is encountered. Spark builds a DAG (a graph where nodes represent RDDs and

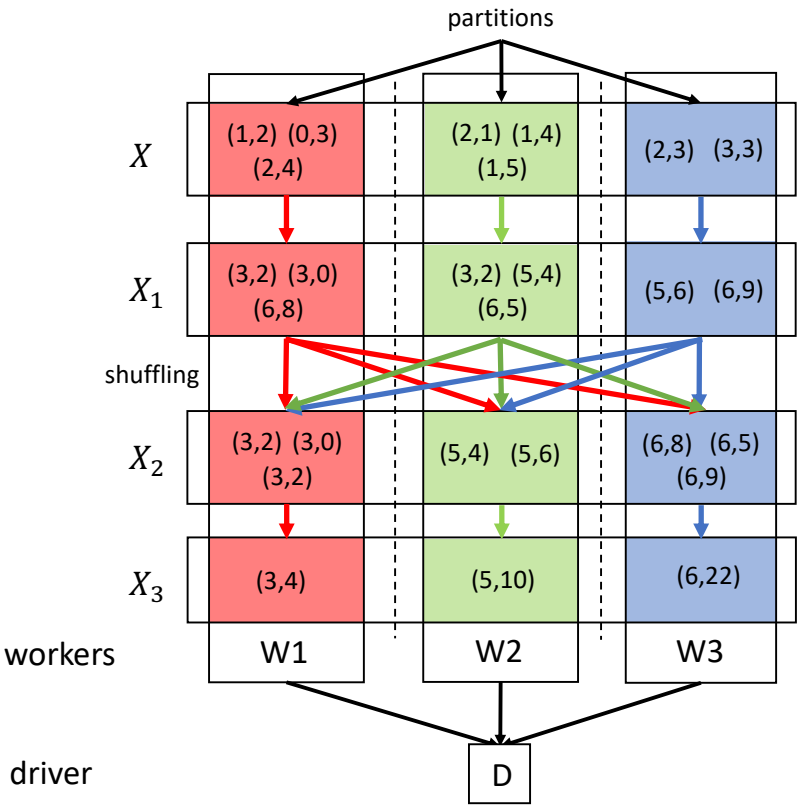


Figure 1.10: Program execution in Spark.

edges represent operations on these RDDs) from all the transformations that occur before an action and schedules it for execution as a job.

Distributed frameworks for graph processing Systems specifically designed for large-scale graph processing include Google's Pregel [Malewicz et al., 2010] and Giraph [gir, 2019] an open-source system based on the Pregel model. GraphX [Gonzalez et al., 2014] is a Spark library for graph processing that offers a Pregel API to perform recursive computations. A more complete survey on these frameworks is found in [Angles and Gutierrez, 2018].

Pregel is based on the vertex-centric model ("Think like a vertex"). A Pregel program is composed of supersteps. At each superstep, a vertex receives messages sent by other vertices at the previous iteration and processes them to update its state and send new messages. Computation stops when no new message is sent. Figure 1.11 shows an example of a Graphx program that computes SSSP: the shortest paths from a source node to every node in the graph. Distance information traverses the graph from node to node, each of which stores it locally and updates it according to the information it receives: At the start, the source node is assigned a distance 0 and the rest of nodes are assigned infinity. At each step, a node sends a distance information to its neighbor (the sum of its distance and the weight of the edge separating it to its neighbor) if this distance is smaller

```scala
import org.apache.spark.graphx.{Graph, VertexId}
import org.apache.spark.graphx.util.GraphGenerators

// A graph with edge attributes containing distances
val graph: Graph[Long, Double] =
  GraphGenerators.logNormalGraph(sc, numVertices = 100).mapEdges(e => e.attr.toDouble)
val sourceId: VertexId = 42 // The ultimate source
// Initialize the graph such that all vertices except the root have distance infinity.
val initialGraph = graph.mapVertices((id, _) =>
    if (id == sourceId) 0.0 else Double.PositiveInfinity)
val sssp = initialGraph.pregel(Double.PositiveInfinity)(
  (id, dist, newDist) => math.min(dist, newDist), // Vertex Program
  triplet => {  // Send Message
    if (triplet.srcAttr + triplet.attr < triplet.dstAttr) {
      Iterator((triplet.dstId, triplet.srcAttr + triplet.attr))
    } else {
      Iterator.empty
    }
  },
  (a, b) => math.min(a, b) // Merge Message
)
println(sssp.vertices.collect.mkString("\n"))
```

Figure 1.11: SSSP algorithm in Graphx [?].

than the one the neighbor already has. Nodes then keep the smallest distance they have received so far. Computation stops when no message is sent.

Is is not straightforward to evaluate UCRPQs in Pregel. An automaton-like algorithm needs to be written to know which stage of the regular query each processed path has reached. The idea is to traverse the paths in the graph (by sending messages from vertices to their neighbors) while traversing the regular query.

1.3.2 Query languages

Distributed frameworks provide an API for users to write applications that target their framework. Such APIs can be considered as internal DSLs. The Spark API for instance is available in Scala, Python, and Java and provides operations to manipulate datasets such as join, groupby, and union as well as second order operations like map and reduce to transform datasets with user defined functions written in the host language. Fig. 1.12 shows a program that multiplies two matrices X and Y using the Spark API. This approach allows for expressing more

```
1   X.map({case (x,i,k) => (k,(x,i))})
2       .join(Y.map({case (y,l,j) => (l,(y,j))}))
3       .map({case (k,((x,i),(y,j))) => ((i,j),x*y)}).reduceByKey(_+_)
4       .map({case ((i,j),z)} => (z,i,j))
```

Figure 1.12: Spark program example.

complex general purpose computations on complex data and reduces impedance mismatch as user defined types and functions written in the host language can be used directly in the DSL. However, as argued in [Alexandrov et al., 2019], it is hard to optimize automatically because of the limited program context available in the intermediate representation of the DSL. For instance, arguments to second order operations are treated as black box functions which means that they cannot be analyzed and transformed to make automatic optimizations.

Other internal DSLs targeting distributed computation frameworks rely on *quotations* to embed their syntax into the host language: DSL terms are delimited by quotes. Examples of such DSLs are DIQL [Fegaras and Noor, 2018] and Emma [Alexandrov et al., 2019]. These languages use Scala macros as quotes. Fig. 1.14 and Fig. 1.13 show the matrice multiplication program written in each of DIQL and Emma. The macros that delimit the queries are shown in green in the figures. The advantage of this approach is the ability to use reflection

```
1  q("""
2      SELECT (+/z, i, j)
3      FROM (x, i, k) <- X, (y,l,j) <- Y, z = x*y
4      WHERE k==l
5      GROUP BY (i,j)
6  """)
```

Figure 1.13: DIQL example.

```
1  onSpark{
2      val pdts = for {(x,i,k) <- X; (y,l,j) <- Y; if(k == l); z = x*y)} yield
           ((i,j),z)
3      for {Group((i,j),zs) <- pdts.groupby(_._1)} yield (zs.fold(Sum),i,j)
4  }
```

Figure 1.14: Emma example.

features of the host language in order to analyse host language terms in the DSL, perform typechecking, access the compiler's symbol table, etc. This facilitates automatic optimizations of the DSL.

External DSLs for distributed frameworks include Spark SQL which enable the user to write SQL queries and represent relational data using the Spark `Dataset` and `DataFrame`. The Pig [Olston et al., 2008] query language targets Hadoop MapReduce. Hive [Thusoo et al., 2009] exposes HiveQL, an SQL-like query language that can run on top of Hadoop or Spark. While they allow for more automatic optimizations they bring back the problem of impedance mismatch and a limited expressivity regarding types and UDFs that can be used in the language as well as other features like recursion and query nesting.

1.4 Conclusion

In this section, we have seen that numerous solutions are available for relational data querying, graph querying, and distributed evaluation. Each present a number of advantages and drawbacks regarding a number of aspects. In order for a program written in a given DSL to get executed, it first gets translated by the compiler to an intermediate representation (IR), then transformed to executable code. At the IR stage, it can undergo a process of analysis and optimization to obtain a code that achieves better performances. A suitable IR for a DSL is one that is capable of representing its programs and enables their optimization. The goal of this work is to discuss and explore formal approaches to intermediate representations that tackle the following issues: expressing recursion, automatic optimization, large-scale data processing and distribution, graph processing, reducing impedance mismatch and querying complex data. We will tackle literature on this topic in the next chapter.

Chapter 2

Formal foundations

2.1 Introduction

In order to optimize a DSL, the DSL compiler translates it to an intermediate representation (or algebra) that gets optimized by means of automatic transformations. These transformations are done through *rewrite rules* that take a query representation (algebraic expression) and produce an equivalent, yet more efficient, one.

Relational Algebra (RA) for instance is an intermediate representation that allows for representing relational data (composed of rows that are each composed of columns) and operations on them such as joins and filters. It is an algebra that has benefited from decades of research, in particular on algebraic rewrite rules for query optimization. Pushing down filters (which reduce the size of their input) as close as possible to the queried datasets is an example of such a rewrite rule. Datalog and its magic sets technique is also another prevalent approach to query optimization. In Datalog, a program is composed of rules that are expressed on predicates, and each predicate depends on a set of variables (which is another representation of relational data).

However, for more complex computations on more complex data like nested collections, using RA as a formalism for the DSL requires flattening the data and using ad-hoc solutions for supporting UDFs. This means that: (1) at the language level, we could have a query language expressed on a flat data model which causes impedance mismatch issues. Alternatively, we could have a query language like LINQ (LINQ to SQL more specifically) where care has to be taken for generating from an embedded query expressed on complex objects a number of SQL queries expressed on data in their flattened form, as well as to generate a flat version of the original query result type. Techniques like O/R (object relational) mappers which necessitate annotations from the user and the implementation of complex techniques like loop lifting and avalanche safety [Giorgidze et al., 2011] are required to guide this process. Additionally, restrictions are made on host language expressions that are allowed in user defined functions due to the fact that it is hard to transform them to relational variants. (2) at the algebraic level, a number of additional joins are introduced to go from hierarchical to flat types and vice versa which has an impact on performance. Additionally, arguments

to second order operations are treated as black box functions which means that they cannot be analyzed and transformed to make automatic optimizations. It is then important to investigate intermediate representations for expressing and optimizing queries in their native format.

As argued by Meijer in [Meijer and Bierman, 2011], establishing and standardizing a formal background for the noSQL market, which now contains multiple separate systems and solutions, is necessary for its economic growth as it was the case for the SQL market thanks to the introduction of RA. The author considers that an algebra based on *monads* is a suitable formalism for this purpose. Emma and DIQL presented in section 1.3.2 are embedded DSLs that can express programs on complex data with minimized impedance mismatch with the host language they are embedded in. Emma is based on *Algebraic Data Types* extended to *monads*, and DIQL is based on the *monoid algebra* as a formal background. Both these formalisms allow for representing collections of any host language type and defining operations on them such as group by and join, as well as second order operations that take a UDF as an argument.

In the next sections, we will discuss each of these formalisms in more detail.

2.2 Relational algebra

The relational model is based on n-ary relations to represent entities and relationships between them. It was introduced by Codd [Codd, 1970] who proposed the idea of a separation between the internal representation (physical storage) from the logical representation of data. The idea is to offer a level of abstraction to represent data and operate upon it via a universal language which is independent from implementation details and possible changes to how data is physically stored and retrieved. This insight led to the relational model and the associated relational algebra being widely adopted by database systems and extensively studied in database research. It also led to the standard SQL language.

2.2.1 Relations

Intuitively, an n-ary relation can be seen as a table of n columns with a header and a body. The header consists of a set of n attributes and the body consists of a set of rows. For a given row r, the value r_i at the position i corresponds to the value of the attribute a_i (at the position i in the header) for that row. There exist slightly different formal definitions of a relation in the literature. In particular, we find two definitions with different emphasis on the importance of attribute names [Abiteboul et al., 1995]. The first approach defines a relation as a set of n-tuples (tuples of n elements). Here, the order of the elements in a tuple and the arity of the relation are important but not the column names. The second approach defines a relation as a set of mappings (functions). A mapping takes an attribute name and returns a value (see definition 2).

Relational algebra defines a set of operations that manipulate relations and derive other relations. These operations constitute a declarative language in which a query (algebraic term) is specified by composing database relations and

operations. From this query specification, the system finds a good strategy to evaluate it. For this, relational algebra uses rewrite rules to transform a term into an equivalent one that is more efficient. In section 2.2.2, we give a description of the RA operations and in section 2.2.3, we give examples of these rules.

2.2.2 Operations

Projection π_A A is a comma separated list of attributes. The relation $\pi_A(\varphi)$ has A as attributes and corresponds to the relation φ from which all attributes have been removed except those in A. The set of attributes in A must be a subset of φ attributes.

Filter $\sigma_\mathfrak{f}$ $\mathfrak{f}$ is a boolean expression that compares attributes to values. For instance, the filter expression $att = 5$ states that the attribute att must be equal to 5. $\sigma_\mathfrak{f}(\varphi)$ corresponds to the relation φ that is filtered by keeping only the rows that satisfy $\mathfrak{f}$. The attributes that appear in $\mathfrak{f}$ must all be attributes of φ.

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

Union $\cup$ $\varphi_1 \cup \varphi_2$ corresponds to the relation having as rows the union of the rows of φ_1 and those of φ_2. φ_1 and φ_2 must have the same attributes.

Join $\bowtie$ The relation $\varphi_1 \bowtie \varphi_2$ has as attributes the union of φ_1 and φ_2 attributes. It corresponds to the relation that, when projected to the attributes of φ_1, gives φ_1 and when projected on the attributes of φ_2, gives φ_2.

Antijoin $\triangleright$ The relation $\varphi_1 \triangleright \varphi_2$ is a subset of φ_1. It is obtained by removing from φ_1 all the rows that can be joined with φ_2. In other words, these rows are such that, when projected on the set of attributes C that are common between φ_1 and φ_2, they are found in $\pi_C(\varphi_2)$ the projection of φ_2 on C.

2.2.3 Rewrite rules

Commuting filters

$$\sigma_{f_1}(\sigma_{f_2}(\varphi)) = \sigma_{f_1}(\sigma_{f_2}(\varphi))$$

Commuting joins

$$\varphi_1 \bowtie \varphi_2 = \varphi_2 \bowtie \varphi_1$$

Commuting unions

$$\varphi_1 \cup \varphi_2 = \varphi_2 \cup \varphi_1$$

Changing the order of join applications

$$\varphi_1 \bowtie (\varphi_2 \bowtie \varphi_3) = (\varphi_1 \bowtie \varphi_2) \bowtie \varphi_3$$

Changing the order of union applications

$$\varphi_1 \cup (\varphi_2 \cup \varphi_3) = (\varphi_1 \cup \varphi_2) \cup \varphi_3$$

Changing the order of antijoin applications

$$\varphi_1 \triangleright (\varphi_2 \triangleright \varphi_3) = (\varphi_1 \triangleright \varphi_2) \triangleright \varphi_3$$

Pushing filter into join

$$\sigma_{\mathfrak{f}}(\varphi_1 \bowtie \varphi_2) = \sigma_{\mathfrak{f}}(\varphi_1) \bowtie \varphi_2$$

if all the attributes that appear in $\mathfrak{f}$ are attributes of φ_1.

Pushing projection into join

$$\pi_A(\varphi_1 \bowtie \varphi_2) = \pi_{A \cap att(\varphi_1)}(\varphi_1) \bowtie \pi_{A \cap att(\varphi_2)}(\varphi_2)$$

if $att(\varphi_1) \cap att(\varphi_2) \subset A$, where $att(\varphi)$ denotes the set of attributes of φ

Pushing projection into union

$$\pi_A(\varphi_1 \cup \varphi_2) = \pi_A(\varphi_1) \cup \pi_A(\varphi_2)$$

Pushing filter into antijoin

$$\sigma_{\mathfrak{f}}(\varphi_1 \triangleright \varphi_2) = \sigma_{\mathfrak{f}}(\varphi_1) \triangleright \varphi_2$$

and

$$\sigma_{\mathfrak{f}}(\varphi_1 \triangleright \varphi_2) = \sigma_{\mathfrak{f}}(\varphi_1) \triangleright \sigma_{\mathfrak{f}}(\varphi_2)$$

if all the attributes that appear in $\mathfrak{f}$ appear in φ_2.

For some of these rules, like pushing filters and projections, it is clear that their application improves performance. The reason is that a filter or a projection operation has a linear complexity, so it is not as costly as other operations such as join and the size of its output is generally smaller than the size of its input. However, this is unclear regarding the application of other rules such as changing the order of join applications. To make decisions, optimizers use heuristics that are computed statically or dynamically.

2.3 Relational Algebra with recursion

2.3.1 μ-RA algebra

The μ-RA algebra [Jachiet et al., 2020] is an extension of the Codd's relational algebra with a recursive operator whose aim is to support recursive terms and enable their transformation to efficient variants. We first present the μ-RA data model, then we give an overview on its syntax, semantics, and the rewrite rules it introduces.

μ-RA data model

Like in the relational model, data in μ-RA consists of relations. A Relation is formally defined as a set of mappings. Let $\mathfrak{V}$, $\mathfrak{C}$, $\mathfrak{R}$ be infinite sets which represent values, column names and relation names respectively.

Definition 1. *A mapping is a function* $m\colon \mathfrak{C} \to \mathfrak{V}$ *having a finite domain denoted* $dom(m)$.

For a mapping m with $dom(m) = \{c_1, ..., c_n\}$, $c_1, ..., c_n$ are called *columns* or *attributes*. They are the values for which m is defined. The set notation $\{c_1 \to m(c_1), ..., c_n \to m(c_n)\}$ is sometimes used to denote the mapping m.

Definition 2. *A relation is a finite set of mappings having the same domain (called the type of the relation).*

Example 1. *Let us consider a directed and rooted graph G as represented in Fig. 2.1. E is a relation that represents the edges in G, and S is a relation that represents starting edges (a subset of edges in E that start from the graph root nodes). The relation S has two columns and contains 4 mappings, one of which is the mapping* $\{ src \to 1, dst \to 2 \}$.

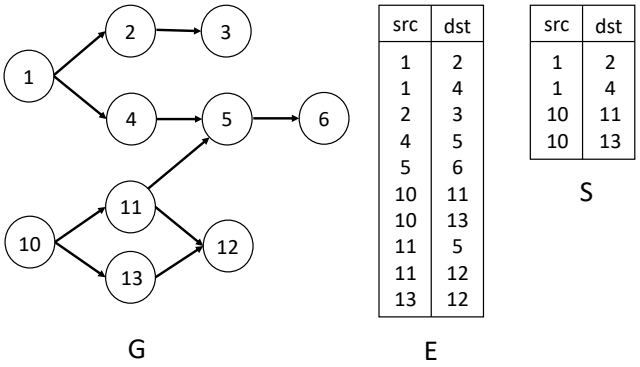


Figure 2.1: Graph example.

μ-RA syntax

The syntax of μ-RA is shown in Fig. 2.2. It is composed of database relation variables and operations (like join and filter) that are applied on relations and yield other relations. These operations correspond to the RA operations described in Sec. 2.2.2 with the additional fixpoint operator μ. In $\mu(X = \Psi)$, X is called *the recursive variable* of the fixpoint term.

Let us consider the graph and relations of Fig. 2.1. The following examples illustrate how μ-RA algebraic terms can be used to model graph operations, such as navigating through a sequence of edges in a graph:

φ	::=		term
		X	relation variable
	$\mid$	$\lvert c \to v \rvert$	constant
	$\mid$	$\varphi_1 \cup \varphi_2$	union
	$\mid$	$\varphi_1 \bowtie \varphi_2$	natural join
	$\mid$	$\varphi_1 \triangleright \varphi_2$	antijoin
	$\mid$	$\sigma_{\mathfrak{f}}(\varphi)$	filtering
	$\mid$	$\rho_a^b(\varphi)$	renaming
	$\mid$	$\widetilde{\pi}_a(\varphi)$	anti-projection (column dropping)
	$\mid$	$\mu(X = \Psi)$	fixpoint term

Figure 2.2: Grammar of μ-RA [Jachiet et al., 2020].

Example 2. *The term $\widetilde{\pi}_c(\rho_{dst}^c(S) \bowtie \rho_{src}^c(E))$ returns pairs of nodes that are connected by a path of length 2 where the first element of the pair is a graph root node. For that purpose, the relation S is joined ($\bowtie$) with the relation E on the common column c, after proper renaming (ρ) to ensure that c represents both the target node of S and the source node of E. After the join, the column c is discarded by the anti-projection ($\widetilde{\pi}_c$) so as to keep only the two columns `src`, `dst` in the result relation.*

Example 3. *Now, the recursive term $\mu(X = S \cup \widetilde{\pi}_c(\rho_{dst}^c(X) \bowtie \rho_{src}^c(E)))$ computes the pairs of nodes that are connected by a path in G starting from edges in S.*

The subterm $\varphi = \widetilde{\pi}_c(\rho_{dst}^c(X) \bowtie \rho_{src}^c(E))$ computes new paths by joining X (the previous paths) and E such that the destinations of X are equal to the sources of E.

The fixpoint is computed in 4 steps where X_i denotes the value of the recursive variable at step i:

$$
\begin{aligned}
X_0 &= \emptyset \\
X_1 &= \Big\{\{src \to 1, dst \to 2\}, \{src \to 1, dst \to 4\}, \\
&\qquad \{src \to 10, dst \to 11\}, \{src \to 10, dst \to 13\}\Big\} \\
X_2 &= X_1 \cup \Big\{\{src \to 1, dst \to 3\}, \{src \to 1, dst \to 5\}, \\
&\qquad \{src \to 10, dst \to 5\}, \{src \to 10, dst \to 12\}\Big\} \\
X_3 &= X_2 \cup \Big\{\{src \to 1, dst \to 6\}, \{src \to 10, dst \to 6\}\Big\} \\
X_4 &= X_3 \quad \textit{(fixpoint reached)}
\end{aligned}
$$

At step 1 it is empty, at step 2 it is a relation of two columns `src` and `dst` that contains four rows, and the iteration continues until the fixpoint is reached.

Semantics and properties of the fixpoint

The semantics of a μ-RA term is defined by the relation obtained after substituting the free variables in the term (like E and S in example 3) by their corresponding

database relations. The notions of free and bound variables and substitution are formally defined in [Jachiet et al., 2020].

As a slight abuse of notation, we sometimes use a recursive term Ψ (i.e. a term that contains a recursive variable X) as a function $R \to \Psi(R)$ that takes a relation R and returns the relation obtained by replacing X in the term Ψ by the relation R. In the above example

$$\varphi(S) = \widetilde{\pi}_c(\rho^c_{dst}(S) \bowtie \rho^c_{src}(E)) = \Big\{\{src \to 1, dst \to 3\}, \{src \to 1, dst \to 5\}, \\ \{src \to 10, dst \to 5\}, \{src \to 10, dst \to 12\}\Big\}$$

Under this notation, $\mu(X = \Psi)$ is defined as the fixpoint F of the function Ψ, so $\Psi(F) = F$.

Let us consider the following conditions (denoted F_{cond}) for a fixpoint term $\mu(X = \Psi)$.

- *positive*: for all subterms $\varphi_1 \triangleright \varphi_2$ of Ψ, φ_2 is constant in X (i.e. X does not appear in φ_2);
- *linear*: for all subterms of Ψ of the form $\varphi_1 \bowtie \varphi_2$ or $\varphi_1 \triangleright \varphi_2$, either φ_1 or φ_2 is constant in X;
- *non mutually recursive*: when there exists a subterm $\mu(Y = \psi)$ in Ψ, then any occurence of X in this subterm should be inside a term of the form $\mu(X = \gamma)$.

These conditions guarantee the following properties:

Proposition 1. *If $\mu(X = \Psi)$ satisfies F_{cond} then*

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

and thus Ψ has a fixpoint with $\mu(X = \Psi) = \Psi^\infty(\emptyset)$.

For instance, $\mu(X = R \triangleright X)$ is not positive, $\mu(X = X \ltimes X)$ is not linear, and $\mu(X = \mu(Y = \varphi(X)))$ is mutually recursive. Whereas $\mu(X = R \cup X \ltimes \mu(Y = \varphi(Y)))$ satisfies F_{cond}.

Proposition 2. *Every fixpoint term $\mu(X = \Psi)$ that satisfies F_{cond} can be written like the following: $\mu(X = R \cup \varphi)$ where R is constant in X and $\varphi(\emptyset) = \emptyset$. R is called **the constant part** of the fixpoint and φ **the variable part**.*

In Example 3, S is the constant part and $\widetilde{\pi}_c(\rho^c_{dst}(X) \bowtie \rho^c_{src}(E))$ is the variable part.

Rewrite rules

We first present some useful definitions before presenting the rewrite rules.

Definition 3. *The **set of derivations** of a term φ, denoted $d(\varphi, X)$, is a set of functions that map column names in φ to other column names. It is defined by the following rules:*

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

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

Intuitively, this notion describes the dependence relationships between columns in the term φ and the columns of X. For instance, $d(\rho_b^a(X) \cup \rho_c^a(X), X) = \{\{a \to b\}, \{a \to c\}\}$ which means that the column a depends on the columns b and c of X.

Definition 4. *Given a term* φ *linear and positive in a variable* X*, we define the* ***stabilizer*** *of* X *in* φ *as the following set of column names:*

$$stab(\varphi, X) = \{c \in \mathfrak{C} \mid \forall p \in d(\varphi, X) \quad p(c) = c\}$$

Intuitively, $stab(\varphi, X)$ is the set of columns that are untouched by the computation of φ.

Definition 5. *We say that a column* c *can be* added to *or removed from a term* ψ *recursive in* X *when* $add(\psi, X, c) = \top$ *holds, with add defined as:*

$$
\begin{array}{lcl}
add(\varphi_1 \cup \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\
add(\varphi_1 \bowtie \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\
add(\varphi_1 \triangleright \varphi_2, X, c) & = & add(\varphi_1, X, c) \wedge add(\varphi_2, X, c) \\
add(\rho_a^b(\varphi), X, c) & = & add(\varphi, X, c) \wedge c \notin \{a, b\} \\
add(\widetilde{\pi}_a(\varphi), X, c) & = & add(\varphi, X, c) \ \textit{when } c \neq a \\
add(\widetilde{\pi}_c(\varphi), X, c) & = & X \notin free(\varphi) \\
add(\sigma_f(\varphi), X, c) & = & add(\varphi, X, c) \wedge f \text{ does not depend on } c \\
add(\mu(Y = \varphi), X, c) & = & add(\varphi, X, c) \\
add(R, X, c) & = & c \notin type(R) \quad \textit{when } X \neq R \\
add(X, X, c) & = & \top \\
add(|c' \to v|, X, c) & = & c \neq c'
\end{array}
$$

Pushing a filter into a fixpoint

$$\sigma_{\mathfrak{f}}(\mu(X = R \cup \varphi)) \rightsquigarrow \mu(X = \sigma_{\mathfrak{f}}(R) \cup \varphi)$$

This transformation is applicable when the set of columns on which the filter $\mathfrak{f}$ depends is included in $stab(\varphi, X)$.

Pushing a join into a fixpoint

$$\varphi \bowtie \mu(X = R \cup \psi) \rightsquigarrow \mu(X = \varphi \bowtie R \cup \psi)$$

This rule is applicable when:

- $t_\varphi \subset stab(\psi, X)$
- $\forall c \in t_\varphi \setminus t \;\; add(\psi, X, c)$

where t_φ is the type of φ and t the type of $\mu(X = R \cup \psi)$.

Merging fixpoints

$$\mu(X = R_1 \cup \varphi_1) \bowtie \mu(X = R_2 \cup \varphi_2) \rightsquigarrow \mu(X = R_1 \bowtie R_2 \cup \varphi_1 \cup \varphi_2)$$

This rule is applicable when:

- $t_1 \cap t_2 \subseteq stab(\varphi_2, X) \cap stab(\varphi_1, X)$
- $\forall c \in t_1 \setminus t_2 \quad add(\varphi_2, X, c)$
- $\forall c \in t_2 \setminus t_1 \quad add(\varphi_1, X, c)$

where t_1 and t_2 are the types of the fixpoint terms.

Evaluation of the fixpoint

A fixpoint term can be evaluated with the algorithm:

Algorithm 1

```
1 X = R
2 new = R
3 while new ≠ ∅:
4     new = φ(new)\ X
5     X = X ∪ new
6 return X
```

The fixpoint is obtained by evaluating φ repeatedly starting from $X = R$ until the fixpoint is reached. In this algorithm, we apply φ on the new results only (obtained by making a set difference between the current result and the previous one) instead of the entire result set. This is possible thanks to the property of φ stated in proposition 1, which implies that $\varphi(X_i) \cup \varphi(X_{i+1}) = \varphi(X_i) \cup \varphi(X_{i+1} \setminus X_i)$.

Evaluating recursive computations on the new iteration results is well known in the context of Datalog [Abiteboul et al., 1995] and transitive closure evaluation [Ioannidis, 1986] with the semi-naive (or differential) approach.

2.3.2 Other extensions of RA with recursion

Earlier works that extend RA with recursion are either less expressive than μ-RA or are more expressive but support less optimizations than μ-RA. α-extended RA [Agrawal, 1988] introduces an operation α that expresses transitive closure, which means it has the same expressivity as UCRPQs (Sec. 1.2.3). LFP-RA [Aho and Ullman, 1979] introduces a least fixpoint operation and has the same expressivity as Datalog with stratified negation. Various fragments of LFP-RA

have also been studied, notably the fragment that is restricted to linear recursion and is as expressive as linear Datalog. Finally, the WHILE [Abiteboul et al., 1995] language is at least as expressive as LFP-RA. It is shown in [Jachiet et al., 2020] that μ-RA with the restrictions F_{cond} (Sec. 2.3.1) is as expressive as linear Datalog. So μ-RA is more expressive than α-extended RA, as expressive as the linear version of LFP-RA, and less expressive than the other formalisms. Several other works introducing recursion in RA can be found in the literature. We present here the closest to μ-RA. Complete surveys are found in [Abiteboul et al., 1995] and [Bancilhon and Ramakrishnan, 1986].

Regarding optimizations, [Aho and Ullman, 1979] proposes a fragment of LFP-RA where some optimisations for recursion like pushing filters inside the fixpoint can be applied, but the formalism is more restricted than μ-RA and cannot express a merged fixpoints term for instance. In its unrestricted form, LFP-RA can be optimized as proposed in [Kifer and Lozinskii, 1990] but μ-RA provides more optimizations that would be incorrect for terms without the F_{cond} restrictions presented earlier.

2.4 Datalog

Datalog is a declarative programming language. A program in Datalog is a collection of *rules* as shown in the example below:

```
1  ancestor(A,C) :- ancestor(B,C), parent(A,B)
2  ancestor(A,B) :- parent(A,B)
3  query(A) :- ancestor(A, "Elisabeth II")
```

This program is composed of three rules. `ancestor(A,C)` in the first rule is called the *head* of the rule. `ancestor(B,C), parent(A,B)` is called the *body* of the rule. It is composed of a conjunction of two *predicates*: `ancestor(B,C)` and `parent(A,B)`. The predicate `parent(A,B)` depends on two *variables*: `A` and `B`, while the predicate `ancestor(A, "Elisabeth II")` depends on a *constant* `"Elisabeth II"` and a variable `A`. The first rule can be intuitively interepreted as: if `A` is a `parent` of `B` and `B` is an `ancestor` of `C`, then `A` is an `ancestor` of `C`. This rule is recursive because it defines the predicate `ancestor` in terms of itself. Non recursive Datalog has the same expressivity as relational algebra [Abiteboul et al., 1995]. Datalog has also a relational data model as predicates can be seen as relations. A database instance I is composed of facts (predicates with constant arguments) such as `parent("Georges VI","Elisabeth II")`. One way to evaluate a Datalog program P is to start from the facts in I and infer new facts from the rules of P. For instance, if `parent("Georges VI","Elisabeth II")` $\in I$, then we can infer that `ancestor("Georges VI","Elisabeth II)`. This fact is called an *immediate consequence* of P and I. We can then update our database instance by adding this fact and infer new facts that are immediate consequences of P and the updated I until no new fact is inferred. More formally, the semantics of a Datalog program can be defined as the least fixpoint of the *immediate consequence operator* [Abiteboul et al., 1995].

The existence of this fixpoint is guaranteed in Linear Datalog and stratified datalog with negation, which are two important fragments among the various fragments of Datalog that have been studied in the literature [Abiteboul et al., 1995].

2.4.1 Datalog optimizations

Datalog can be seen as a formalism to reason about program optimization. Magic Sets [Bancilhon et al., 1986, Saccà and Zaniolo, 1986], recently improved upon by Demand Transformation [Tekle and Liu, 2011], is a major line of work on Datalog optimization. These techniques analyze the rules in a datalog program and transform them to produce an optimized equivalent datalog program. The core idea of the optimization is to produce only the facts that are relevant to a query. Often, a query restricts the value of a predicate argument to a constant. The query of the program above for instance asks for the ancestors of `"Elisabeth II"`. If we follow the execution described earlier, from a fact `parent("Georges V", "Georges VI")` in I, the execution will generate `ancestor("Georges V", "Georges VI")` that is going to be eliminated by the query rule. Magic Sets uses adornment which is a technique that consists in adding, in the program, information about which predicate variables are restricted (bound variables) and which are not (free variables). For instance, an adornment of the program above leads to the following program ($\texttt{ancestor}^{fb}\texttt{(A,C)}$ means that `A` is free and `C` is bound):

```
1  ancestor^fb(A,C) :- ancestor^fb(B,C), parent(A,B)
2  ancestor^fb(A,B) :- parent(A,B)
3  query(A) :- ancestor(A, "Elisabeth II")
```

Adornments can intuitively be seen as variables transmitting their values from the query to the predicates that need to be computed. Transmission is done from the query rule to the heads other rules, from the head of a rule to its body, and from left to right in the body of a rule. In the first rule of the above example, `C` is bound in the head of the rule because it is set to a constant in the query, and it is bound in $\texttt{ancestor}^{fb}\texttt{(B,C)}$ because it is bound in the head of the rule. Note that adornment is omitted on the `parent` predicate because it is an *EDB predicate* (a source predicate in the database, not derived by the program). Otherwise, it would have the adornment $\texttt{parent}^{fb}(A, B)$ because `B` is also an argument of `ancestor` to its left, so its value is transmitted from `ancestor` to `parent`. From the adorned program, new predicates (called magic predicates) that represent constraints on the bound variables are defined and inserted as additional conjuncts in rule bodies, thus restraining the facts generated by these rules. In our example, this leads to the following equivalent program:

```
1  ancestor^fb(A,C) :- magic_ancestor^f(C), ancestor^fb(B,C), parent(A,B)
2  ancestor^fb(A,B) :- magic_ancestor^f(C), parent(A,B)
3  magic_ancestor^f("Elisabeth II") :-
4  query(A) :- ancestor(A, "Elisabeth II")
```

Instead of computing the entire `ancestor` predicate, this program only computes the ancestors of `"Elisabeth II"`.

This optimization technique however depends on how the original program is written. If, for instance, the first rule of the program was written in the following way

```
ancestor(A,C) :- ancestor(A,B), parent(B,C)
```

this would lead to the following adorned rule

$$\texttt{ancestor}^{fb}\texttt{(A,C) :- ancestor}^{ff}\texttt{(A,B), parent(B,C)}$$

where both arguments of `ancestor` are free. This means that the entire `ancestor` is going to be computed. The rule of the original program recursively defines `ancestor` as being the parent of an ancestor, while this rule defines it as being the ancestor of a parent. A datalog program with the first form of recursion is called a *right-linear* program, while a program with the second form of recursion is called a *left-linear* program. [Naughton et al., 1989] proposes a *reversal* technique that enables transforming one of these types of program into the other. Magic Sets is also sensitive to the order of predicates in the rule bodies. If we simply swap the predicates in the body of the first rule of the example, a less efficient program (that performs more operations) will be produced even though only the ancestors of `"Elisabeth II"` get computed. Another performance consideration is the cost of computing magic predicates with respect to the selectivity they provide. A bad strategy can even degrade the performance of the original program as observed in [Green et al., 2013]. Works in [Sereni et al., 2008] and [Seshadri et al., 1996] investigate these issues using runtime and static approaches.

2.4.2 Comparison between μ-RA and Magic sets for optimizing recursion

The optimizations provided by Magic Sets are equivalent to pushing selections and projections in μ-RA. However, there is no equivalent to merging fixpoints. As mentioned earlier, depending on the way the Datalog program is written, some optimizations may or may not be applied. For instance a left-linear DL program (e.g. $P(x, y) \leftarrow P(x, z), R(z, y)$) cannot push filters that are applied on the right side (on y in the example). In order to account for all possible filters that can be pushed into recursion, Magic Sets has to be coupled with a technique for reversing DL programs. Since Datalog engines use heuristics to combine optimization techniques, optimizations are not always performed as observed in [Jachiet et al., 2020].

2.4.3 Distributed systems based on Datalog

Recent works that studied the distribution of Datalog programs are SociaLite [Seo et al., 2013], which is an extension of Datalog for social network analysis with graphs. They implement a distributed system that runs queries on a cluster of multi-core machines in which workers communicate using message passing. Myria [Wang et al., 2015] is a distributed system that supports a subset

of Datalog extended with aggregation. Queries are translated into query plans that are executed on a parallel relational engine. Myria supports incremental evaluation of recursion and provides both a synchronous and an asynchronous mode for evaluating recursive queries. BigDatalog [Shkapsky et al., 2016] is a recursive Datalog engine that runs on Spark. Datalog programs are translated into Spark physical plans. They propose optimizations that aim at reducing Spark shuffle operators from the physical plans, hence reducing communications among the workers. They also propose SetRDD, which is an optimized specialization of the Spark RDD (Sec. 1.3.1) for sets.

2.5 Collection algebra

In the relational model, data consists of relations that are sets of tuples of atomic values. A more generic data model are collections of arbitrary homogeneous types. There are 3 well-known types of collections: lists, bags, and sets. They are, along with other types of collections, part of what is called the Boom hierarchy of types [Bunkenburg, 1993]. The author of this paper presents collections (or data structures as he calls them) as free algebras. A collection value can either be empty [ ], a singleton $[a]$, or a concatenation of two values $c1 +\!\!+ c2$. $+\!\!+$ can obey to a combination of the following algebraic laws:

(1) unit: $a +\!\!+ [\,] = a = [\,] +\!\!+ a$

(2) associativity: $a +\!\!+ (b +\!\!+ c) = (a +\!\!+ b) +\!\!+ c$

(3) commutativity: $a +\!\!+ b = b +\!\!+ a$

(4) idempotence: $a +\!\!+ a = a$

Different combinations of laws lead to different types of collections. [Bunkenburg, 1993] defines 16 types of collections for all possible combinations of these laws. For instance, *tree* is a collection type where only the first law is satisfied. If we consider $A = \{1, 2, 3\}$, elements of tree[A] are all the possible nodes that can be constructed in a binary tree having [1],[2], and [3] as leaves. Fig. 2.3 shows a part of this tree. Note that $(([1] +\!\!+ [2]) +\!\!+ [3])$ and $(([1] +\!\!+ ([2] +\!\!+ [3])$ are different elements of $tree[A]$, which is not the case for $list[A]$ because the concatenation operator of *list* satisfies associativity (in addition to unit). If we add commutativity we obtain bags, and if we add idempotence we obtain sets. Collections with a concatenation operator satisfying (1) and (2) (which is the case for lists, bags, and sets) are monoids. Fegaras [Fegaras, 2017] calls them *collection monoids*. It is on this basic notion that he builds the *monoid algebra*. Collections can also be seen as a particular case of Algebraic Data Types which constitutes the basic notion of the Emma language (Sec. 1.3.2) approach. Both of these approaches propose an algebra for distributed collections. We will next see each of them in more detail.

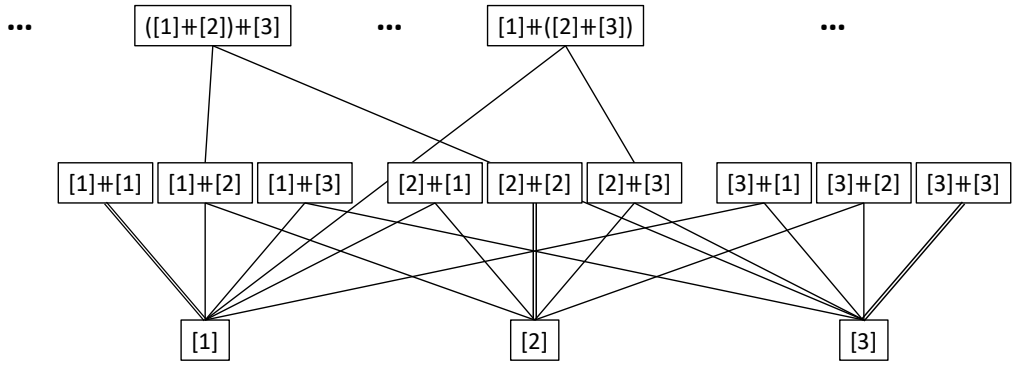


Figure 2.3: Tree example.

2.5.1 Algebraic Data Types (Emma approach)

The Emma language [Alexandrov et al., 2019] presented in Sec.1.3.2 is based on the following algebraic approach:

- distributed collections are defined as Algebraic Data Types.
- operations on collections are defined as *fold* operations.
- collections are extended to a *monad with zero* which can be queried using *for comprehensions*.

Definition of collections

An Algebraic Data Type (ADT) is a composite type that is built by combining other types through union (sum types) or product (tuples) or other type constructors.

Lists are defined as a parametric type $List[A]$ that corresponds to the following ADT:

$$List[A] = emp \mid sng\ A \mid uni\ List[A]\ List[A]$$

where *emp* is the empty list constructor, *sng* is a constructor that takes an element in A and builds the singleton containing this element, and *uni* is a constructor that takes two lists of A elements and builds their union. Constructors can be associated with axioms that they must satisfy. For instance the *uni* constructor of lists satisfies the following axioms:

$$uni\ l\ emp = uni\ emp\ l = l$$
$$uni\ (uni\ l_1\ l_2)\ l_3 = uni\ l_1\ (uni\ l_2\ l_3)$$

These two laws correspond to the unit and associativity laws presented earlier. So, other types of collections can be defined the same way as lists, but with different set of axioms associated to their *uni* constructor. The *uni* constructor of bags for instance satifies the following axiom in addtition to the two axioms above: $uni\ l_1\ l_2 = uni\ l_2\ l_1$.

The author notes that this union representation (using *uni*) of collections is suitable for representing distributed collections that are partitioned among different nodes.

Operations on collections

Every ADT can be transformed with a *fold* operation. This operation is parametrized by functions that are associated to the constructors of the ADT and that describe the computation to perform for each component of the ADT. For example, the *fold* operation of lists is the recursive function defined as follows:

$$fold(zero, init, plus) = f : List[A] \to B$$
$$f(l) = \begin{cases} zero & \text{if } l = emp \\ init(x) & \text{if } l = sng(x) \\ plus(f(l_1), f(l_2)) & \text{if } l = uni(l_1, l_2) \end{cases}$$

where $zero : B$, $init : List[A] \to B$, and $plus : B \times B \to B$.

The *fold* operation in this work represents the basic primitive of distributed collection processing. Computing *fold* on a list that is split to different partitions can be obtained by applying the *fold* operation on each partition in parallel, then by aggregating the obtained results using *plus*. In addition, it is a second order operation with polymorphic types, which allows it to accept generic user defined functions, and can be used to define more specific operations like groupby.

A collection ADT can be extended to a *monad with zero*. It is an algebraic structure consisting of a tuple $(emp, sng, map, flatten)$, where *emp* and *sng* are the collection constuctors, and *map* and $flatten$ are functions that can be defined using *fold* in the following way:

$$map(f : A \to B) : Coll[A] \to Coll[B] = fold(emp, sng \circ f, uni)$$
$$flatten : Coll[Coll[A]] \to Coll[B] = fold(emp, id, uni)$$

where *Coll* is used to denote one of Bag, List, or Set.

We can perform monad comprehensions on monads. Monad comprehensions offer a declarative syntax to query collections. For example

$$\texttt{for}\ \{p \leftarrow Person;\ a \leftarrow Address;\ \texttt{if}\ p.address\_id = a.id\ \}\ \texttt{yield}\ (p.name, a.city)$$

is a comprehension that computes the names of people and the city they live in. The expression $(p.name, a.city)$ is the *head* of the comprehension and the expression between brakets is the *qualifier* sequence of the comprehension. A qualifier can either be (1) a generator like $p \leftarrow Person$ which binds each element of $Person$ to the variable p or (2) a *guard* like $p.address\_id = a.id$ which is a boolean expression.

Rewrite rules

[Alexandrov et al., 2019] applies a number of rewrite rules to compile for comprehensions into the monad with zero interface (map, withFilter, flatMap). We present two important comprenhension compilation rules then we present other rewrite rules involving fold operations.

Pushing filters in comprehensions

$$\texttt{for } \{qs_1;\ \ x \leftarrow xs;\ \ qs_2;\ \ \texttt{if } p;\ \ qs_3\} \texttt{ yield } let$$
$$\rightsquigarrow \texttt{for } \{qs_1;\ \ x \leftarrow x_s.\texttt{withFilter}(x \Rightarrow p);\ \ qs_2;\ \ qs_3\} \texttt{ yield } let$$

This rule is applicable when the guard p references the variable x and does not reference any bound variable (that appears in the left side of a generator) in qs_1 and qs_2.

Extracting joins from comprehensions

$$\texttt{for } \{qs_1;\ \ x \leftarrow xs;\ \ qs_2;\ \ y \leftarrow ys;\ \ qs_3;\ \ \texttt{if } k_x = k_y;\ \ qs_4\} \texttt{ yield } let$$
$$\rightsquigarrow \texttt{for } \{qs_1;\ \ (x, y) \leftarrow \texttt{equiJoin}(x \Rightarrow k_x, y \Rightarrow k_y)(xs, y_s);\ \ qs_2;\ \ qs_3;\ \ qs_4\} \texttt{ yield } let$$

This transformation is applicable when:

- k_x references x and does not reference y and the variables that are bound in qs_1, qs_2, and qs_3
- k_y references y and does not reference x and the variables that are bound in qs_1, qs_2, and qs_3
- y does not depend on x, meaning that x does not appear in y_s, nor in any generator that binds variables in y_s, nor in the generators that bind variables in them, and so on.

Equijoin is defined as a function and has different implementations for the different platforms on which the query can be executed. The Spark implementation for instance calls the Spark join operator.

Cata-fusion This rule fuses a fold operation and a map operation into one fold operation:

$$fold(zero_1, init_1, plus_1)(map(f)(X)) \rightsquigarrow fold(zero_2, init_2, plus_2)(X)$$

where

$$zero_2 = zero_1$$
$$init_2 = init_1 \circ f$$
$$plus_2 = plus_1$$

Banana fusion This rewrite rule consists in fusing multiple folds on the same dataset into a single fold:

$(fold(zero_1, init_1, plus_1)(X), fold(zero_2, init_2, plus_2)(X)) \rightsquigarrow fold(zero_3, init_3, plus_3)(X)$

where

$zero_3 = (zero_1, zero_2)$

$init_3 = x \rightarrow (init_1(x), init_2(x))$

$plus_3 = ((x_1, x_2), (y_1, y_2)) \rightarrow (plus_1(x_1, y_1), plus_2(x_2, y_2))$

So in the Emma language, an expression of the form

```
a = X.fold(alg1)
b = X.fold(alg2)
```

would be rewritten to

```
f = X.fold(Alg(alg1,alg2)))
a = f._1
b = f._2
```

Fold-group fusion In Emma, `groupby` is a method of the collection ADT that is defined as a special fold. It takes a $Coll[A]$ and a key function `kf` : $A \rightarrow K$ and returns a $Coll[Group(K, A)]$ such that elements in the input collection having the same key are grouped in the output. The fold fusion rule allows to fuse this groupby operation with a generic fold operation performed on the groups returned by groupby:

```
val gb = X.groupby(kf)
val res = for (Group(k,group) <- gb) yield {
    ... // alg definition instructions
    val f = group.fold(alg)
    ...
}
```

would be rewritten to

```
    ... // alg definition instructions
    val fg = X.foldGroup(kf,alg)
    val res = for (Group(k,f) <- fg) yield {
        ...
    }
```

where `foldGroup` is a special fold operation that, given a key function `kf` and a fold operation `alg`, groups its input by key and folds each group using `alg`.

2.5.2 Monoid algebra approach

Definition of collections

The algebra proposed by Fegaras [Fegaras, 2017, Fegaras and Noor, 2018] is based on the notion of *monoids*. A monoid $(S, \oplus, e)$ is an algebraic structure where S

is a set (called the carrier set of the monoid), $\oplus$ an associative binary operator between elements of S, and e is an identity element for $\oplus$. $\oplus$ can satisfy additional laws like commutativity and idempotence. The author notes that collections such as lists, bags, and sets have the structure of a monoid whose binary operator is the associative concatenation operator. He terms them *collection monoids*. Given an arbitrary type α, a collection monoid is a monoid equipped with a unit injection function $\mathcal{U}_{\otimes} : \alpha \to T(\alpha)$ that maps an element of type α to a singleton (a collection of type $T(\alpha)$ containing this element only). The main collection monoids are presented in the table below:

	$\otimes$	$T(\alpha)$	e	$\mathcal{U}_{\otimes}(x)$	properties
list	$+\!\!+$	$List[\alpha]$	$[\,]$	$[x]$	
bag	$\uplus$	$Bag[\alpha]$	$\{\{\,\}\}$	$\{\{x\}\}$	commutative
bag	$\cup$	$Set[\alpha]$	$\{\,\}$	$\{x\}$	commutative amd idempotent

Table 2.1: Collection monoids [Fegaras and Noor, 2018]

Collection monoids are suitable for representing distributed collections. Associativity means that the whole collection can be seen as the union of the subcollections stored on the different machines without specifying an order in which to apply the union operator.

Operations on collections

A *monoid homomorphism* h from $(S, \oplus, e)$ to $(S', \otimes, e')$ is a function $h : S \to S'$ such that $h(x \oplus y) = h(x) \otimes h(y)$ and $h(e) = e$.

Collection monoids, as they are free monoids, satisty the following universal property:

Proposition 3. *let $(T(\alpha), \oplus, e_{\oplus})$ be a collection monoid and $(\beta, \otimes, e_{\otimes})$ a monoid, where α and β are arbitrary types. Suppose $\otimes$ obeys all the algebraic laws of $\oplus$, then for any function $f : \alpha \to \beta$, there exists a unique homomorphism $H_f^{\otimes} : T(\alpha) \to \beta$ such that $H_f^{\otimes} \circ \mathcal{U}_{\otimes} = f$.*

In the case of bags for instance, this property means for that the bag homomorphism $H_f^{\otimes}$ satisfies the following:

$$H_f^{\otimes}(\{\!\{x\}\!\}) = f(x)$$
$$H_f^{\otimes}(\{\!\{\}\!\}) = e_{\otimes}$$
$$H_f^{\otimes}(X \uplus Y) = H_f^{\otimes}(X) \otimes H_f^{\otimes}(Y)$$

For example, given the monoid $(\texttt{Int}, +, 0)$ and the function $\texttt{one} : x \to 1$, we have that $H^{+}_{\texttt{one}}$ is the monoid homomorphism which counts the elements of its input bag. Here we can see that the associativity property of $\uplus$ and $\otimes$ is interesting in the context of distributed programming because it is possible to compute $H_f^{\otimes}$(X) by dividing X into multiple parts, applying the computation on each part independently, then gathering the results using the $\otimes$ operator without leading to erroneous results.

As mentioned earlier, in order for the universal property to be satisfied, $\otimes$ must satisfy all the laws of $\oplus$. In our example above, $H^{+}_{\mathtt{one}}$ cannot be defined for sets because + is not idempotent. We have $H^{+}_{\mathtt{one}}(\{1,2,3\}) = 6$ and $H^{+}_{\mathtt{one}}(\{1,2,1,3\}) = 7$, while the sets $\{1,2,3\}$ and $\{1,2,1,3\}$ are equal, which is contradictory.

A homomorphism $H^{\otimes}_{f}$ having a collection monoid as a source monoid is referred to as a *collection homomorphism*. It is the basic notion that captures operations on collections in the monoid algebra. Examples of such operations are second order operators which are parametrized by an arbitrary function f that manipulates collection elements or an arbitrary binary operator $\otimes$ that combines them. Homomorphisms are equivalent to the $fold$ operator of the ADT approach presented earlier. An ADT defining a collection forms a monoid over its polymorphic type because its constructor satisfies the associativity and the unit axiom, and we have $fold(zero, init, plus) = H^{plus}_{init}$. We present below the collection homomorphisms that are defined in the monoid algebra proposed by Fegaras [Fegaras and Noor, 2018].

flatmap operator We consider a function $f : \alpha \to \mathsf{Bag}[\beta]$. $\mathsf{flmap}(f,\, X)$ applies f on each element in the bag X and returns a dataset that is the union of all results. This operation is a monoid homorphism $H^{\uplus}_{f}$ from $(\mathsf{Bag}[\alpha], \uplus, \{\!\{\}\!\})$ to $(\mathsf{Bag}[\beta], \uplus, \{\!\{\}\!\})$.

reduce operator $\mathsf{reduce}(\oplus,\, e_{\oplus},\, X)$ reduces the elements of the input dataset by combining them with the $\oplus$ operator, which must be associative and commutative. This operation is a monoid homomorphism $H^{\oplus}_{id}$ from $(\mathsf{Bag}[\alpha], \uplus, \{\!\{\}\!\})$ to $(\mathsf{Bag}[\beta], \oplus, e_{\oplus})$.

groupby operator groupby takes a bag of elements in the form (k, v), where k is considered the *key* and v the *value*, and returns a bag of elements in the form (k, V) where V is the bag of all elements having the same key in the input dataset. Thus, each key appears exactly once in the result. For example, $\mathsf{groupby}(\{(1,2),(1,4),(2,2),(2,1),(1,3)\}) = \{(1,\{2,4,3\}),(2,\{2,1\})\}$.

groupby is a monoid homorphism $H^{\uparrow}_{f}$ from $(\mathsf{Bag}[\alpha \times \beta], \cup, \{\!\{\}\!\})$ to $(\mathsf{Bag}[\alpha \times \mathsf{Bag}[\beta]], \uparrow, \{\!\{\}\!\})$, where:

$$f\colon \alpha \times \beta \to \mathsf{Bag}[\alpha \times \mathsf{Bag}[\beta]]$$
$$(k, v) \mapsto \{\!\{(k, \{\!\{v\}\!\})\}\!\}$$

and the operator $\uparrow$ is defined such that

$$\{\!\{(k, b_1)\}\!\} \uparrow \{\!\{(k', b_2)\}\!\} = \begin{cases} \{\!\{(k, b_1 \uplus b_2)\}\!\} & \text{if } k = k' \\ \{\!\{(k, b_1), (k', b_2)\}\!\} & \text{otherwise} \end{cases}$$

cogroup operator cogroup takes two collections of elements of the form (k, v) and (k, w) and returns a collection of elements of the form $(k, (V, W))$ where V and W are the sets of v values and w values having the same key k.

cogroup is a binary homomorphism $H^{\updownarrow}_{f_1,f_2}$ defined in the following way:

$$\begin{aligned}\mathsf{cogroup}(X_1 \uplus Y_1, X_2 \uplus Y_2) &= \mathsf{cogroup}(X_1, Y_1) \updownarrow \mathsf{cogroup}(X_2, Y_2)\\ \mathsf{cogroup}(\{\!\{x\}\!\}, \{\!\{y\}\!\}) &= f_1(x) \updownarrow f_2(y)\end{aligned}$$

where:

$$\begin{aligned}f_1 \colon \alpha \times \beta &\to \mathsf{Bag}[\alpha \times (\mathsf{Bag}[\beta]) \times \mathsf{Bag}[\gamma])]\\ (k, v) &\mapsto \{\!\{(k, (\{\!\{v\}\!\}, \{\!\}))\}\!\}\\ f_2 \colon \alpha \times \gamma &\to \mathsf{Bag}[\alpha \times (\mathsf{Bag}[\beta] \times \mathsf{Bag}[\gamma])]\\ (k, v) &\mapsto \{\!\{(k, (\{\!\}, \{\!\{v\}\!\}))\}\!\}\end{aligned}$$

and: $\{\!\{(k, (b_1, c_1))\}\!\} \updownarrow \{\!\{(k', (b_2, c_2))\}\!\} = \begin{cases}\{\!\{(k, (b_1 \uplus b_2, c_1 \uplus c_2))\}\!\} & \text{if } k = k'\\ \{\!\{(k, (b_1, c_1)), (k', (b_2, c_2))\}\!\} & \text{otherwise}\end{cases}$

Rewrite rules

In this section, we present important rewrite rules that have been proposed for the monoid algebra.

Note: In the monoid algebra, we use the lambda expression notation $\lambda\, \langle x \to e \rangle$ to denote a function that takes an argument x and returns e. The argument of a lambda expression can also be a pattern. For instance, $\lambda\, \langle (x, y) \to x \rangle$ is a function that takes a tuple and returns its first element. For a formal definition of lambda expressions, patterns and pattern matching please refer to Sec. 4.2.

Pushing filter into groupby

$$\begin{aligned}&\mathsf{flmap}(\lambda\, \langle (k, xs) \to g(\mathsf{flmap}(\lambda\, \langle x \to \mathsf{if}\ c\ \mathsf{then}\ e\ \mathsf{else}\ \{\!\{\}\!\} \rangle, xs)) \rangle,\\ &\qquad \mathsf{groupby}(X))\\ \rightsquigarrow\ &\mathsf{flmap}(\lambda\, \langle (k, xs) \to g(\mathsf{flmap}(\lambda\, \langle x \to \mathsf{if}\ c_2\ \mathsf{then}\ e\ \mathsf{else}\ \{\!\{\}\!\} \rangle, xs)) \rangle,\\ &\qquad \mathsf{groupby}(\mathsf{flmap}(\lambda\, \langle x \to \mathsf{if}\ c_1\ \mathsf{then}\ x\ \mathsf{else}\ \{\!\{\}\!\} \rangle, X)))\end{aligned}$$

where c is a boolean term and g is a term function (a term in which the argument appears as a subterm) such that $g(\{\!\{\}\!\}) = \{\!\{\}\!\}$. In the first expression, the groups returned by $\mathsf{groupby}(X)$ are traversed and their elements are filtered using the predicate c. This transformation consists in prefiltering X before applying groupby on it. It is applicable when the predicate c can be split to two predicates c_1 and c_2 such that $c = c_1 \wedge c_2$ and c_1 may contain the k and x variables while c_2 must not.

Converting nested flatmaps to joins Let us consider the following term

$$F(X, Y) = \mathsf{flmap}(\lambda\, \langle x \to g(\mathsf{flmap}(\lambda\, \langle y \to \mathsf{if}\ p(x, y)\ \mathsf{then}\ e\ \mathsf{else}\ \{\!\{\}\!\} \rangle, Y)) \rangle, X)$$

where g and p are term functions. This term consists of nested flatmaps that traverse the datasets X and Y, and for each pair x and y of their elements return a result if the predicate $p(x, y)$ is true and nothing otherwise. In order to evaluate

this term on Spark, it is necessary to broadcast the Y dataset on the workers. This can be avoided if we apply the following transformation rule:

$$\begin{aligned} F(X,Y) \rightsquigarrow \mathsf{flmap}(\lambda\ \langle (k,(xs,ys)) \rightarrow F(xs,ys)\rangle, \\ \mathsf{cogroup}(\mathsf{flmap}(\lambda\ \langle x \rightarrow \{\!\{(k_1(x),x)\}\!\}\rangle, X), \\ \mathsf{flmap}(\lambda\ \langle y \rightarrow \{\!\{(k_2(y),y)\}\!\}\rangle, Y))) \end{aligned}$$

where k_1 and k_2 are key functions. This transformation is applicable when we can derive k_1 and k_2 from the predicate $p(x,y)$ such that $k_1(x) \neq k_2(y) \Rightarrow \neg p(x,y)$. This condition means that if the pairs x and y for which $k_1(x) \neq k_2(y)$ are not joined in the cogroup operation, no result will be lost. A straightforward example of such predicate is $k_1(x) = k_2(y)$.

Elimination of groupby inside cogroup When a cogroup is applied on a groupby and its join key is the same as the groupby key, then the groupby can be eliminated like the following:

$$\begin{aligned} &\mathsf{cogroup}(\mathsf{flmap}(\lambda\ \langle (k,s) \rightarrow \{\!\{(k,f(s))\}\!\}\rangle, \mathsf{groupby}(X)), Y) \\ &\rightsquigarrow \mathsf{flmap}(\lambda\ \langle (k,(xs,ys)) \rightarrow \{\!\{(k,(\{\!\{f(xs)\}\!\},ys))\}\!\}\rangle, \mathsf{cogroup}(X,Y,)) \end{aligned}$$

Converting cogroup into groupby

$$\begin{aligned} &\mathsf{cogroup}(\mathsf{flmap}(f,X), \mathsf{flmap}(g,X)) \\ &\rightsquigarrow \mathsf{flmap}(\lambda\ \langle (k,s) \rightarrow (k,(\texttt{retainLeft(s)},\texttt{retainRight(s)}))\rangle, \\ &\qquad \mathsf{groupby}(\mathsf{flmap}(\lambda\ \langle x \rightarrow \mathsf{flmap}(\lambda\ \langle (k,a) \rightarrow (k,\texttt{left}(a))\rangle, f(x)) \\ &\qquad\qquad \uplus\, \mathsf{flmap}(\lambda\ \langle (k,b) \rightarrow (k,\texttt{right}(b))\rangle, g(x))\rangle, \\ &\qquad X))) \end{aligned}$$

where `left` and `right` are functions that are used to distinguish the left argument from the right argument of the cogroup operator. `retainLeft` and `retainRight` are functions that filter their input to keep only the elements that are marked as left and right respectively. They can be defined by the following:

$$\begin{aligned} &\texttt{left(a)} = (a,1) \\ &\texttt{right(a)} = (a,2) \\ &\texttt{retainLeft(s)} = \mathsf{flmap}(\lambda\ \langle (x,t) \rightarrow \mathsf{if}\ t == 1\ \mathsf{then}\ x\ \mathsf{else}\ \{\!\{\}\!\}\rangle, s) \\ &\texttt{retainRight(s)} = \mathsf{flmap}(\lambda\ \langle (x,t) \rightarrow \mathsf{if}\ t == 2\ \mathsf{then}\ x\ \mathsf{else}\ \{\!\{\}\!\}\rangle, s) \end{aligned}$$

This transformation is useful for the distributed evaluation on Spark because the groupby operation requires less synchronization overhead than the cogroup operation.

2.5.3 Nested RA

Previous works have studied the extension of the RA model to support complex values. In these models, a relation is allowed to contain values that can also be

relations. A *complex value relation* [Abiteboul and Beeri, 1995] is defined as a set of complex values, where a complex value can be a set of complex values ($\{v_1, ..., v_n\}$), a named tuple of complex values ($\langle A_1 : v_1, ..., A_n : v_n \rangle$), or an atomic value. Types of collections other than sets can also be defined. *The nested relation model* is a more restricted version of the complex value model: A nested relation is a set of tuples of values that can either be atomic or nested relations. The complex value algebra [Abiteboul and Beeri, 1995] defines the following core operations on complex value relations:

- $\cup$, $\cap$, $\setminus$ binary set operators
- $select\langle p \rangle(R)$ filters R with p, an arbitrary boolean-valued function
- $replace\langle f \rangle(R)$ returns the relation obtained after applying an arbitrary function f on each element of R
- $powerset(R)$ returns the powerset of its input relation (which is a set)
- $cross_{A_1,...,A_n}(R_1, ..., R_n)$ builds a relation that has A_1, ..., A_n as columns. The values of each column A_i range over the elements of R_i.
- *set-collapse*(R) takes a set of sets and flattens it into one set.

In summary, the complex value data model defines complex values using the set constructor and the named tuple constructor as well as atomic type values. The algebra offers a second order *replace* operator that accepts an arbitrary (well-typed) function and that is used to transform relation elements, but no reduce or fold operator is proposed. In comparison, the approaches presented earlier define the data model as collections of elements that can be defined using arbitrary type constructors. Algebraic operators like map and reduce are formalized using generic concepts such as the monoid homomorphism or fold concepts. The two approches are then very similar, but the second is more appropriate as a destination of internal DSLs.

2.6 Conclusion

In this chapter, we have seen formalisms based on the relational model like RA and Datalog as well as extensions of RA to support recursion. These approaches are fairly similar but present some variations in terms of expressivity and proposed optimizations. Next, we have seen approaches that are based on a more generic data model: collections of arbitrary homogenous types. In this model, data can be nested and various types of collections such as sets, lists, and bags can be used. Operations on collections are defined using generic concepts like monads and monoid homomorphisms. Second order operators can accept arbitrary user defined functions and thus offer a more flexible way of manipulating complex data. Some automatic rewrite rules are also proposed.

In the next chapters, we explore extensions of the formal approaches along these two lines of work. In chapter 3, we present Dist-μ-RA, an extension of μ-RA

to the distributed setting. In chapter 4, we present μ-monoids, an extension of the monoid algebra with a fixpoint operator.

Part II

Contribution

Introduction

In the first part of this manuscript we have explored differents facets of the following question: how to query and manipulate data efficiently?

In chapter 1, we have seen a variety of languages and tools to query data of various forms: graphs, relational data, and large distributed data. Some of them, like SQL, are easy to use but suffer from the impedance mismatch problem. Others, like the Spark API, are well integrated with their host language but require manual tuning and expertise to achieve good performances. Languages like Emma or DIQL aim to provide automatic optimizations while reducing impedance mismatch.

In order to provide automatic optimizations and generate a concrete execution plan, these languages are translated to intermediate represention. We have seen in chapter 2 formalisms based on the relational model like relational algebra and Datalog. When it comes to querying relational data, we can say that RA (and Datalog, possibly to a lesser extent) is the formalism of choice because of the large body of research it has benefited from over the years. We will see that for querying edge-labelled graphs (which can be represented as a relation having src, label, target as columns) a formalism based on RA performs better/has access to more optimizations than current tools specifically made for querying graphs. While pure RA lacks recursion, Datalog can naturally express recursion by having the same predicate in the head and body of one of its rules. μ-RA is a formalism that extends RA with recursion capabilities. Compared to recursive Datalog and other formalisms that extend RA with recursion, μ-RA presents advantages in terms expressivity and optimizations. However, it is limited to the centralized setting. We study in chapter 3 the distribution of μ-RA terms.

For expressing distributed programs that handle complex data, we have seen formalisms that allow for modelling distributed collections and expressing operations on them. The Emma approach uses Algebraic Data Types (ADT) to represent collections of any host language type. These ADTs are extended to a monad that is equipped with structural recursion (fold operation) which allows for representing operations on collections. The DIQL approach is based on the monoid algebra as a formal background. Collection monoids are used to represent collections of any host language type and monoid homomorphisms are used to define operations, such as group by and join, on these collections. Both fold operations and monoid homomorphisms allow for defining second order operations that take a UDF as an argument. However, regarding the ability to express recursion, the Emma formalism lacks a recursion operator and uses host language loops (not captured by its algebra) to express iterative computations. While the DIQL formalism has a repeat operator which is not homomorphic and is not subject to algebraic optimizations. We study in chapter 4 the addition of a fixpoint operator in the monoid algebra.

Chapter 3

Distributed evaluation of recursive relational queries: Dist-μ-RA

3.1 Introduction

Several works have addressed the problem of query optimization in the presence of recursion, in particular with extensions of Relational Algebra [Agrawal, 1988, Aho and Ullman, 1979, Jachiet et al., 2020]; and with Datalog-based approaches [Abiteboul et al., 1995] such as BigDatalog [Shkapsky et al., 2016]. Recently, μ-RA [Jachiet et al., 2020] proposed logical optimization rules for recursion not supported by earlier approaches. In particular, these rules include the merging and reversal of recursions that cannot be done neither with Magic sets nor with Demand Transformations that constitute the core of optimizations in Datalog-based systems [Jachiet et al., 2020]. However, μ-RA is limited to the centralized setting.

In this chapter, we present Dist-μ-RA a new method and its implementation for the optimized distributed evaluation of recursive relational algebra terms. Specifically, the novelty is twofold:

1. a new method for the optimization of distributed evaluation of queries written in recursive relational algebra. Since it uses a general recursive relational algebra, it can be of interest for a large number of mainstream RDBMS implementations; and it can also provide the support for distributed evaluation of recursive graph query languages. For example Dist-μ-RA provides a frontend where the programmer can formulate queries known as UCRPQs [Consens and Mendelzon, 1990, Barcelo et al., 2012, Barceló et al., 2012, Libkin et al., 2016][1]).

   This method provides a systematic parallelisation technique by means of physical plan generation and selection. These plans automatically repar-

[1] UCRPQs, discussed in more details in Sec. 3.2, constitute an important fragment of expressive graph query languages: they correspond to unions of conjunctions of regular path queries. A translation of UCRPQs into the recursive relational algebra is given in [Jachiet et al., 2020].

tition data in order to reduce data transfer between cluster nodes and communication costs during recursive computations.

2. a prototype implementation [dis, 2022] of the system on top of Apache Spark. Specifically, Dist-μ-RA can use plain Apache Spark or Apache Spark with PostgreSQL as a DBMS backend. To evaluate Dist-μ-RA experimentally, a classification of graph queries by the means of seven query classes has been defined. Each class characterizes queries with a particular feature: for example a recursion with a filter, or concatenated recursions. Dist-μ-RA is evaluated using queries that cover the different classes, and using datasets (both real and synthetic) of various sizes. Experimental results show that Dist-μ-RA is more efficient than state-of-the-art systems such as BigDatalog [Shkapsky et al., 2016] in most query classes.

The outline of the chapter is as follows: we first describe the architecture of Dist-μ-RA in 3.2. In Section 3.3, we show how μ-RA terms are distributed and how physical plans are generated. Finally, we report on experimental evaluation in Section 4.4 and related works in Section 3.5 before concluding.

3.2 Dist-μ-RA architecture

The Dist-μ-RA system takes a query as input parameter, translates it into μ-RA, optimizes it, and then performs the evaluation in a distributed fashion on top of Spark. Specifically, the Dist-μ-RA system is composed of several components, as illustrated in Fig. 3.1.

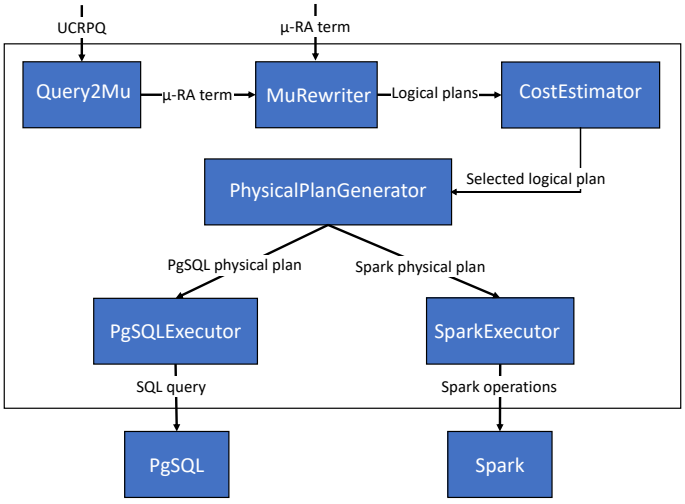


Figure 3.1: Architecture of the Dist-μ-RA system.

The `Query2Mu` component translates recursive graph queries written in Union of Conjunctive Regular Path Queries (UCRPQ) into μ-RA terms. The UCRPQ syntax is given in Sec. 1.2.3 and we give an example below. Dist-μ-RA supports

more general μ-RA terms that are not expressible as UCRPQs[2], as long as they satisfy the triple condition F_{cond} mentioned in Section 2.3.1.

From a given input μ-RA term, the `MuRewriter` explores the space of semantically equivalent logical plans by applying a number of rewrite rules. In addition to the rewrite rules already known in classical relational algebra, `MuRewriter` applies a set of rules specific to the fixpoint operator. These rules and the conditions under which they are applicable are formally defined in [Jachiet et al., 2020].

The evaluation costs of these terms are estimated by the `CostEstimator` component, based on cardinality estimations proposed in [Lawal et al., 2020]. Based on these estimations, a best logical recursive plan is selected.

From a given recursive logical plan, the `PhysicalPlanGenerator` generates a physical plan for distributed execution (see Section 3.3). Two distributed execution setups are used. In the first setup (using `PgSQLExecutor`), each Spark worker runs a PostgreSQL instance to perform a part of the evaluation locally. The second setup (using `SparkExecutor`) relies only on Spark. In all cases, the query evaluation is performed on top of Spark.

Example We first describe the transformations that a query (UCRPQ or μ-RA term) undergoes before being considered for distributed evaluation when given as input parameter to the `PhysicalPlanGenerator` (described in Section 3.3).

Consider for instance the following UCRPQ composed of a conjunction of two Regular Path Queries (RPQs):

```
?a,?b,?c ←?a wasBornIn/IsLocatedIn+ Japan,
          ?b isConnectedTo+ ?c
```

The first RPQ computes the people `?a` that are born in a place that is located directly or indirectly in `Japan`. The query is first translated into μ-RA by `Query2Mu` so that `MuRewriter` can generate semantically equivalent plans. We describe below the rewrite rules specific to fixpoint terms leveraged from [Jachiet et al., 2020] and presented in Sec. 2.3.1 that can apply in `MuRewriter`, and we give the intuition of their effect on performance:

- *Pushing filters into fixpoints*: with this rule, the query `?x isLocatedIn+ Japan` is evaluated as a fixpoint starting from `?x` such as `?x isLocatedIn Japan`, which avoids the computation of the whole `isLocatedIn+` relation followed by the filter `Japan`.
- *Pushing joins into fixpoints*: let us consider the query `?x isMarriedTo/knows+ ?y`. Instead of computing the relation `knows+` and joining it with `isMarriedTo`, this rule rewrites the fixpoint such that it starts from `?x` and `?y` that verify `?x isMarriedTo/knows ?y`. The application of this rule is beneficial in this case because the size of the `isMarriedTo/knows` relation is usually smaller than the size of the knows relation.

[2]See the practical experiments section for some examples such as the "same generation" query.

- *Merging fixpoints*: when evaluating `?x isLocatedIn+/dealsWith+ ?y`, instead of computing both fixpoints separately then joining them, this rule generates a single fixpoint that starts with `isLocated/dealsWith` then recursively appends either `isLocatedIn` to the left or `dealsWith` to the right.

- *Pushing antiprojections into fixpoints*: this rule gets rid of unused columns during the fixpoint computations. For instance, the query `?y` ← `?x isLocatedIn+ ?y` (this query asks for `?y` only) is evaluated by starting only from the destinations `?y` of the `isLocatedIn` relation and by recursively getting the new destinations, thus avoiding to keep the pairs of nodes `?x` and `?y` then discarding `?x` at the end.

- *Reversing a fixpoint*: the fixpoint corresponding to the relation $a+$ can either be computed from left to right by starting from a and by recursively appending a to the right of the previously found results, or from right to left by starting from a and appending a to the left. Reversing a fixpoint consists in rewriting from the first form to the other or vice versa. This rule is necessary to account for all possible filters and joins that can be pushed in a fixpoint. For instance, a filter that is located at the left side of $a+$ can only be pushed if the fixpoint is evaluated from left to right.

After these transformations, the best (estimated) recursive logical plan selected by `CostEstimator` is given as input parameter to `PhysicalPlanGenerator` that is in charge of generating the best physical plan for distributed execution.

3.3 Distributed evaluation

We now describe how fixpoint terms are evaluated in a distributed manner, first by explaining the principles and then how physical plans are generated.

3.3.1 Fixpoint distributed evaluation principles

The first principle uses a global loop on the Spark *driver*[3]. The second principle uses parallel local loops on the Spark workers, and corresponds to our contribution.

Global Loop on the Driver ($\mathcal{P}_{\texttt{gld}}$)

$\mathcal{P}_{\texttt{gld}}$ corresponds to the natural way a Spark programmer would implement the fixpoint operation: it distributes the computations performed at each iteration of Algorithm 1 (Sec. 2.3.1). This execution is illustrated in Fig. 3.2 (left side). Colored arrows show data transfers that occur at each iteration of the fixpoint. The driver performs the loop and, at each iteration, instructions at lines 4 and 5 are executed as `Dataset` [4] operations that are distributed among the workers. We

[3]The *driver* is the process that creates tasks and send them to be executed in parallel by *worker* nodes. See Sec. **??**

[4]Distributed collection data structure used to store relational data in Spark [Armbrust et al., 2015a]

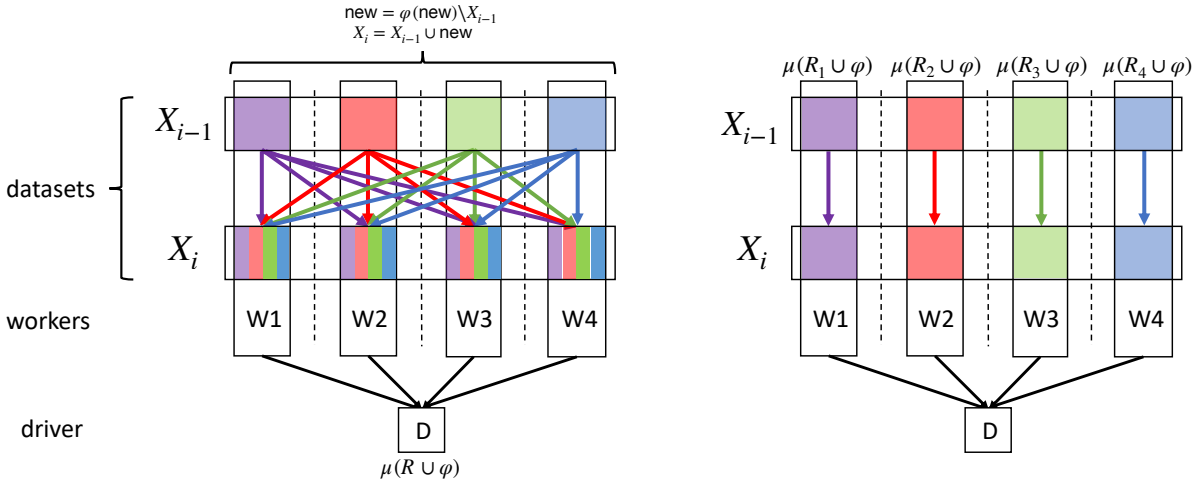


Figure 3.2: Execution on the cluster of $\mathcal{P}_{\texttt{gld}}$ (left) and $\mathcal{P}_{\texttt{plw}}$ (right).

call this execution plan $\mathcal{P}_{\texttt{gld}}$. On Spark, $\cup$ is executed as `Dataset` union followed by a `distinct()` operation. This means that in $\mathcal{P}_{\texttt{gld}}$, at least one data transfer (shuffle) per iteration is made to perform the union.

Parallel Local loops on the Workers ($\mathcal{P}_{\texttt{plw}}$)

This evaluation principle uses the following observation to distribute the fixpoint:

Proposition 4. *Under the conditions F_{cond}, we have:*

$$\mu(X = R_1 \cup R_2 \cup \varphi) = \mu(X = R_1 \cup \varphi) \cup \mu(X = R_2 \cup \varphi)$$

which is a consequence of proposition 1 (Sec. 2.3.1). This proposition means that a fixpoint whose constant part is a union of two datasets can be obtained by making the union of two fixpoints, each with one of these datasets as a constant part. Thanks to this proposition 4, the fixpoint can be executed by distributing the constant part R among the workers, then each worker i executes a smaller fixpoint $\mu(X = R_i \cup \varphi)$ locally starting from its own constant part R_i. We call this execution plan $\mathcal{P}_{\texttt{plw}}$. Execution is illustrated in the right side of Fig. 3.2. As opposed to $\mathcal{P}_{\texttt{gld}}$, $\mathcal{P}_{\texttt{plw}}$ performs only one data shuffle at the end to make the union ($\cup$) between the local fixpoints.

In Example 3, if we split the start edges S among two workers by giving the first $(1, 2)$ and $(10, 11)$ and the second $(1, 4)$ and $(10, 13)$, after executing $\mathcal{P}_{\texttt{plw}}$, the first worker will find the paths $(1, 3)$, $(10, 5)$, $(10, 6)$ and $(10, 12)$ and the second will find $(1, 5)$, $(1, 6)$, and $(10, 12)$.

Data distribution for $\mathcal{P}_{\texttt{plw}}$ There are cases where the final data shuffle induced by $\mathcal{P}_{\texttt{plw}}$ can also be avoided by appropriately repartitioning input data among workers. We first give the intuition behind this idea, followed by the proof.

We look for a column *col* (or a set of columns) in X left unchanged by φ. In other words, a tuple in R having a value v at column *col* will only generate tuples

having the same value at this column throughout the iterations of the fixpoint. So if we put all tuples in R having v at column *col* in one worker, no other worker will generate a tuple with this value at that column.

For this, we use the stabilizer technique defined in Definition 4 (Sec. 2.3.1) and used to push filters in fixpoint expressions. It consists of computing the set of columns which are not altered during the fixpoint iteration. For instance, 'src' is a stable column in the fixpoint expression of example 3 meaning that tuples in the fixpoint having 'src' = 1 can only be produced from tuples in S having 'src' = 1, which implies that filtering tuples having 'src' = 1 before or after the fixpoint computation lead to the same results. However, this is not true for the column 'dst' which is not stable.

To summarize, when the constant part of the fixpoint is repartitioned by the stable column (or columns) prior to the fixpoint execution, we know for certain that there will be no duplicate across the workers (so we can avoid calling `distinct()` at the end of the computation).

For instance, repartitioning the constant part S in example 3 by *src* will result in the paths $(1, 3)$, $(1, 5)$, and $(1, 6)$ being found in one worker and the paths $(10, 5)$, $(10, 6)$ and $(10, 12)$ in the second, thus avoiding a duplicate $(10, 12)$ between the two workers.

Proof 1. *Let c be a stable column of $\mu(X = R \cup \varphi)$, which means that $\forall e \in \mu(X = R \cup \varphi)$ $\exists r \in R$ $e(c) = r(c)$ [Jachiet et al., 2020]. In μ-RA, an element r in R is a mapping (tuple), which means that it is a function that takes a column name and returns the value that r has at that column.*

Let us consider a partitioning R_1, ..., R_n of R by the column c which verifies the following

$$\forall i \neq j \in \{1..n\}\ \forall a \in R_i\ \forall b \in R_j\ \ a(c) \neq b(c)$$

This statement means that there are no two elements of R at different partitions that share the same value at column c. We next show that this statement is also true for the fixpoint term.

Let $i \neq j \in \{1..n\}$ and let $x \in \mu(X = R_i \cup \varphi)$ and $y \in \mu(X = R_j \cup \varphi)$. Since c is stable, we have $\exists a \in R_i$ $x(c) = a(c)$ and $\exists b \in R_j$ $y(c) = b(c)$. So $x(c) \neq y(c)$.

In conclusion, the sets $\mu(X = R_i \cup \varphi)$ where $i \in \{1..n\}$ are disjoint.

3.3.2 Physical plan generation and selection

We present the different physical plans automatically generated by the Dist-μ-RA system for μ-RA terms, and explain how they are selected. Dist-μ-RA generates a physical plan for $\mathcal{P}_{\texttt{gld}}$, which is used only as a baseline in performance comparisons.

We propose two alternative physical plans which are variants of $\mathcal{P}_{\texttt{plw}}$:

- $\mathcal{P}^{\texttt{pg}}_{\texttt{plw}}$: The local fixpoints are executed on PostgreSQL. The fixpoint operator is performed as a Spark `mapPartition()` operation where each worker performs a portion of the fixpoint computation on PostgreSQL. A PostgreSQL instance runs on each worker. The part of data assigned to each worker is represented as a view in the PostgreSQL instance running on this

worker. The μ-RA expression (that computes the fixpoint) is translated to a PostgreSQL query that is executed using this view as the constant part of the fixpoint. The local PostgreSQL plans are selected for the operators in the fixpoint expression. Each PostgreSQL executor returns its results as an iterator which is then processed by Spark.

- $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$: The fixpoint computation is implemented using a loop in the driver that uses Spark operations to compute the recursive part of the fixpoint. These Spark operations are written in such a way that each worker performs its own fixpoint independently (i.e. without data exchanged between workers). Joins are executed as broadcast joins: all relations in the variable part of the fixpoint (apart from the recursive relation) are broadcasted. Antiprojections are executed without the need of applying the `distinct()` operation. To perform the union (or set-difference), a special union (set-difference) operation is used that computes the union (set-difference) partition-wise. These special union and set-difference operations are implemented as part of the `SetRDD` API. `SetRDD` [Shkapsky et al., 2016] is a special `RDD`[5] where each partition is a set. This `SetRDD` is used to store the value of the recursive variable X at each iteration. This means that each partition of X holds the intermediate results of the local fixpoint performed by the worker to which this partition has been assigned.

As a consequence, for each of the non-recursive μ-RA operators, there are two kinds of physical plans: local plans implemented using PostgreSQL and distributed plans implemented using the Spark `Dataset` API. `Dataset`s are used to represent relational data in Spark. The optimization of these expressions is then delegated to Spark's Catalyst internal optimizer [Armbrust et al., 2015b] before execution. Some operators have more than one distributed execution plan. For instance, for the join operator, we choose which argument (if any) to broadcast in order to guide Spark on whether to use broadcast join or another type of join.

To select between the two alternatives $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$ and $\mathcal{P}^{\mathtt{pg}}_{\mathtt{plw}}$, we rely on a simple selection mechanism based on an empirical observation. We have observed that the size of the datasets in the variable part of the fixpoints is the key element that affects the relative performance of the plans. When this size is large, $\mathcal{P}^{\mathtt{pg}}_{\mathtt{plw}}$ is more effective than $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$. We use the cost estimator presented in Sec. 3.2 to determine this size. In practice, when the estimated size exceeds a certain threshold (set to 20M records), we select $\mathcal{P}^{\mathtt{pg}}_{\mathtt{plw}}$ and $\mathcal{P}^{\mathtt{s}}_{\mathtt{plw}}$ otherwise.

3.4 Experiments

We evaluate the performance of a prototype implementation of the Dist-μ-RA system on top of the Spark platform [Zaharia et al., 2016]. We extensively compared its performance against other state-of-the-art systems on various datasets and queries. We report below on these experiments.

[5] `RDD` is an abstraction that Spark provides to represent a distributed collection of data. An `RDD` is split among partitions which are assigned to workers. See Sec. 1.3.1.

3.4.1 Experimental setup

Experiments have been conducted on a Spark cluster composed of 4 machines (hence using 4 workers, one on each machine, and the driver on one of them). Each machine has 40 GB of RAM, 2 Intel Xeon E5-2630 v4 CPUs (2.20 GHz, 20 cores each) and 66 TB of 7200 RPM hard disk drives, running Spark 2.4.5 and Hadoop 2.8.4 inside Debian-based Docker containers.

3.4.2 Datasets

Real Dataset	Description	Edges	Nodes
Yago [for Informatics and University, 2019]	YAGO semantic knowledge base	62,643,951	42,832,856
Epinions [ico, 2022]	Epinions product ratings (2005)	13,668,320	996,744
Wikitree [Fire and Elovici, 2015]	Online genealogy dataset	9,192,212	1,382,751
Coauth-MAG [ico, 2022]	MAG Geology coauthor simplices	5,120,762	1,256,385
Gottron [Kunegis, 2013]	Wikipedia words	2,941,903	273,961
AcTree [Liénard et al., 2018]	Academic Family Tree Data Export	1,561,494	777,220
Wikitree_0 [Fire and Elovici, 2015]	Wikitree filtered on the relation ID 0	1,556,453	1,019,438
Reddit [Leskovec, 2019]	Hyperlinks between subreddits	858,490	55,863
TW-Cannes [ico, 2022]	Cannes 2013 Multiplex social network	991,855	438,539
Higgs-RW [ico, 2022]	Twitter, Higgs boson (2012)	733,647	425,008
Wikidata_c [wik, 2022]	Wikidata child relation	280,405	333,572
Wikidata_p [wik, 2022]	Wikidata father and mother relations	280,740	334,430
Facebook [Leskovec, 2019]	Social circles from Facebook	88,234	4,039
Ragusan [ico, 2022]	Ragusan nobility genealogy	51,938	13,690
Isle-of-Man [ico, 2022]	Isle of Man genealogy (1600-2011)	36,666	10,474
Fr-Royalty [Halliwell et al., 2021]	French royalty genealogy tree	12,358	2,127

Synthetic Dataset	Edges	Nodes
uniprot_10M	10,001,920	10,000,000
uniprot_5M	5,001,427	5,000,000
uniprot_1M	1,000,443	1,000,000
uniprot_100k	66,181	100,000
rnd_100k_0.001	5,003,893	100,000
rnd_10k_0.001	249,791	10,000
rnd_7k_0.001	24,630	7,000
rnd_5k_0.001	12,660	5,000
tree_10k	9,999	10,000
tree_7k	6,999	7,000
tree_5k	4,999	5,000

Table 3.1: Real and synthetic graphs.

We use real and synthetic datasets of different sizes and topological properties, as summarized in Table 4.1. We consider the following real graphs:

- Yago[6]: A knowledge graph extracted mainly from Wikipidia [for Informatics and University, 2019].
- datasets from the Colorado index of complex networks [ico, 2022] and from the Snap network dataset collection [Leskovec, 2019].

In addition, we consider the following synthetic graphs:

- uniprot_n: a benchmark graph of n nodes generated using the gMark benchmark tool [Bagan et al., 2017]. It models the Uniprot database of proteins [Gane et al., 2014].
- rnd_n_p: random graphs generated with the Erdos Renyi algorithm, where n is the number of nodes in the graph and p the probability that two nodes are connected.
- tree_n: a random tree of n nodes generated recursively as follows: tree_1 is a tree of 1 node, and then tree_$i+1$ is a tree of $i+1$ nodes where the $i^{\text{th}}+1$ node is connected as a child of a randomly selected node in tree_i.

3.4.3 Systems

We compare Dist-μ-RA with the following systems:

- BigDatalog [Shkapsky et al., 2016] available at [bdl, 2022]: a large-scale distributed Datalog engine built on top of Spark.
- GraphX [Gonzalez et al., 2014]: a Spark library for graph computations. It exposes the Pregel API for recursive computations. In order to compare our system with GraphX we need to convert UCRPQs to GraphX programs[7]. Specifically, we compute a regular graph query by making each node send a message to its neighbors in such a way that the query pattern is traversed recursively from left to right. This means that for a query that starts by selection `(?x ← A pattern ?x)`, only the node `A` sends a message at the start of the computation.
- Myria [Wang et al., 2015]: a distributed big data management system that supports Datalog and exposes MyriaL, a query language similar to Datalog. We were not able to run this system in a distributed setting because the software stack has evolved and Myria is not maintained anymore. However, we were able to test it on a single machine hosting four workers. In addition, Myria works only with smaller datasets (i.e. the small synthetic graphs presented in Table 4.1) and on a subset of test queries.

[6]We use a cleaned version of the real world dataset Yago 2s, that we have preprocessed in order to remove duplicate RDF [Cyganiak et al., 2014] triples (of the form <source, label, target>) and keep only triples with existing and valid identifiers. After preprocessing, we obtain a table of Yago facts with 83 predicates and 62,643,951 rows (graph edges).

[7]In the GraphX framework, a recursive computation is composed of "supersteps" where, in each superstep, graph nodes send messages to their neighbor nodes, then a merge function aggregates messages per recipient and each recipient receives its aggregated messages in order to process them. A computation is stopped when no new message is sent.

3.4.4 Queries

Queries may contain various forms of recursion. To ensure that tested queries cover different forms of recursion, we rely on a classification of queries in seven classes. Each class regroups queries with a particular recursive feature: $\mathcal{C}_1 - \mathcal{C}_6$ describe UCRPQ queries on knowledge graphs (like Uniprot and Yago). We also provide additional experiments using more general queries not expressible as UCRPQs in the class $\mathcal{C}_7$. The classification is the following:

- $\mathcal{C}_1$ corresponds to queries containing a single transitive closure (TC), e.g. `?x, ?y ← ?x a+ ?y`
- $\mathcal{C}_2$: queries with a filter to the right of a TC, e.g. `?x ← ?x a+ C`
- $\mathcal{C}_3$: queries with a filter to the left a TC, e.g. `?x ← C a+ ?x`
- $\mathcal{C}_4$: queries which contain a concatenation of a non recursive term to the right of a TC, e.g. `?x, ?y ← ?x a+/b ?y`
- $\mathcal{C}_5$: queries which contain a concatenation of a non recursive term to the left of a TC, e.g. `?x, ?y ← ?x b/a+ ?y`
- $\mathcal{C}_6$: queries which contain a concatenation of TCs, e.g. `?x, ?y ← ?x a+/b+ ?y`
- $\mathcal{C}_7$: queries with non regular recursion, e.g. $a^n b^n$.

Each class requires specific optimizations. For instance, the optimization of queries of classes $\mathcal{C}_2$ and $\mathcal{C}_3$ requires pushing filters in fixpoint terms (in two different directions). Queries of classes $\mathcal{C}_4$ and $\mathcal{C}_5$ require an optimization that pushes joins in fixpoint terms. $\mathcal{C}_2$ and $\mathcal{C}_4$ require reversing fixpoint terms before applying other optimizations (rewritings). Queries of $\mathcal{C}_6$ can be optimized by merging fixpoints or by pushing joins in fixpoint terms.

A query may belong to one or more classes. Whenever a query belongs to several classes this means that it requires the optimization techniques of all the corresponding classes, together with a technique capable of combining them. Therefore, the more classes a query belongs to, the harder is its optimization. For example, the query `?x ← C a/b+ ?x` belongs to $\mathcal{C}_3$ because there is a filter to the left of the transitive closure `b+` and also belongs to $\mathcal{C}_5$ because there is a concatenation to the left of `b+`.

To cover a variety of queries in the experiments (see Figures 3.3 and 3.4), there is, for each class $\mathcal{C}_i$, at least one query that belongs to $\mathcal{C}_i$ alone. In addition, we also consider queries that belong to $\mathcal{C}_i$ and to a combination of other classes. This allows to test how the different combinations of optimizations are supported by the tested systems.

Yago queries Fig. 3.3 lists the UCRPQs evaluated on the Yago dataset along with their classes. Queries $\mathcal{Q}_3$ and $\mathcal{Q}_4$ are taken from [Abul-Basher et al., 2017], $\mathcal{Q}_5$ from [Yakovets et al., 2015], and $\mathcal{Q}_6, \mathcal{Q}_7$ from [Gubichev et al., 2013]. We have added queries $\mathcal{Q}_8 - \mathcal{Q}_{25}$ that include larger transitive closures.

$\mathcal{Q}_{id}$	*Query*	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$	$\mathcal{C}_4$	$\mathcal{C}_5$	$\mathcal{C}_6$
$\mathcal{Q}_1$	?x,?y ←?x,?y <- ?x hasChild+ ?y	×					
$\mathcal{Q}_2$	?x,?y ←?x,?y <- ?x isConnectedTo+ ?y	×					
$\mathcal{Q}_3$	?x ←?x isMarriedTo/livesIn/IsL+/dw+ Argentina		×			×	×
$\mathcal{Q}_4$	?x ←?x livesIn/IsL+/dw+ United_States		×			×	×
$\mathcal{Q}_5$	?x ←?x (actedIn/-actedIn)+ Kevin_Bacon		×				
$\mathcal{Q}_6$	?area ←wce -type/(IsL+/dw|dw) ?area			×	×	×	
$\mathcal{Q}_7$	?person ←?person isMarriedTo+/owns/IsL+|owns/IsL+ USA		×		×	×	
$\mathcal{Q}_8$	?x,?y ←?x IsL+/dw+ ?y						×
$\mathcal{Q}_9$	?x,?y ←?x (IsL|dw|rdfs:subClassOf|isConnectedTo)+ ?y	×					
$\mathcal{Q}_{10}$	?x ←?x (isConnectedTo/-isConnectedTo)+ Shannon_Airport		×				
$\mathcal{Q}_{11}$	?person ←?person (wasBornIn/IsL/-wasBornIn)+ JLT		×				
$\mathcal{Q}_{12}$	?x ←Jay_Kappraff (livesIn/IsL/-livesIn)+ ?x			×			
$\mathcal{Q}_{13}$	?x,?y ←?x (actedIn/-actedIn)+/hasChild+ ?y						×
$\mathcal{Q}_{14}$	?x,?y ←?x (wasBornIn/IsL/-wasBornIn)+/isMarriedTo ?y				×		
$\mathcal{Q}_{15}$	?x,?y ←?x (actedIn/-actedIn)+/influences ?y				×		
$\mathcal{Q}_{16}$	?x ←Marie_Curie (hWP/-hWP)+ ?x			×			
$\mathcal{Q}_{17}$	?x ←London -wasBornIn/(playsFor/-playsFor)+ ?x			×		×	
$\mathcal{Q}_{18}$	?x ←London (-wasBornIn/hWP/-hWP/wasBornIn)+ ?x			×			
$\mathcal{Q}_{19}$	?x,?y ←?x -actedIn/(-created/influences/created)+ ?y					×	
$\mathcal{Q}_{20}$	?x,?y ←?x -isLeaderOf/(livesIn/-livesIn)+ ?y					×	
$\mathcal{Q}_{21}$	?x,?y ←?x (-created/created)+/directed ?y				×		
$\mathcal{Q}_{22}$	?x ←Lionel_Messi (playsFor/-playsFor)+/isAffiliatedTo ?y			×	×		
$\mathcal{Q}_{23}$	?x ←SH (haa|influences)+/(isMarriedTo|hasChild)+ ?x			×			×
$\mathcal{Q}_{24}$	?x,?y ←?x isConnectedTo+/IsL+/dw+/owns+ ?y						×
$\mathcal{Q}_{25}$	?x,?y ←?x haa/hasChild/(hWP/-hWP)+ ?y					×	

Figure 3.3: Queries for the YAGO dataset[8].

Concatenated closures We consider queries of the form $a_1+/a_2+/.../a_n+$ where $2 \leq n \leq 10$. These queries all belong to class $\mathcal{C}_6$.

Non regular queries We also consider queries that contain non-regular forms of recursion. These queries are exclusively expressible as μ-RA terms, not as UCRPQs. All of these queries belong to $\mathcal{C}_7$:

- $a^n b^n$ queries: they return the pairs of nodes connected by a path composed of a number of edges labeled a followed by the same number of edges labeled b. They are expressed with the following μ-RA term:

$$\begin{aligned}\mu(X = \widetilde{\pi}_m(\rho^m_{trg}(\sigma_{pred=a}(R)) \bowtie \rho^m_{src}(\sigma_{pred=b}(R)))\\ \cup \widetilde{\pi}_m(\widetilde{\pi}_n(\rho^m_{trg}(\sigma_{pred=a}(R)) \bowtie \rho^n_{trg}(\rho^m_{src}(X))\\ \bowtie \rho^n_{src}(\sigma_{pred=b}(R)))))\end{aligned}$$

- Same Generation (SG) queries: they return the pairs of nodes that are of the same generation in a graph. We use the following term to express

[8] "isL" stands for "IsLocatedIn", "dw" for "dealsWith", "haa" for "hasAcademicAdvisor", "JLT" for "John_Lawrence_Toole", "hWP" for "hasWonPrize", "SH" for "Stephen_Hawking", and "wce" for "wikicat_Capitals_in_Europe".

them:

$$T_{SG} = \mu(X = \widetilde{\pi}_m(\rho^m_{src}(R) \bowtie \rho^m_{src}(R)) \cup \widetilde{\pi}_m(\widetilde{\pi}_n(\rho^m_{src}(R) \bowtie \rho^n_{trg}(\rho^m_{src}(X)) \bowtie \rho^n_{src}(R))))$$

- Filtered SG queries: they compute the pairs of nodes that are of the same generation for a particular predicate p in a graph.

$$\sigma_{pred=p}(T_{SG})$$

- Joined SG: they return the pairs of nodes that are of the same generation for a particular set of predicates P in a graph. P is a one column (*pred*) relation that gets joined with the T_{SG} term on the column *pred*:

$$P \bowtie T_{SG}$$

Uniprot queries For the synthetic Uniprot datasets, we use the UCRPQ queries shown in Fig. 3.4.

$\mathcal{Q}_{id}$	*Query*	$\mathcal{C}_1$	$\mathcal{C}_2$	$\mathcal{C}_3$	$\mathcal{C}_4$	$\mathcal{C}_5$	$\mathcal{C}_6$
$\mathcal{Q}_{26}$	?x,?y ← ?x -hKw/(ref/-ref)+ ?y					×	
$\mathcal{Q}_{27}$	?x,?y ← ?x -hKw/(enc/-enc)+ ?y					×	
$\mathcal{Q}_{28}$	?x ← C (occ/-occ)+ ?x			×			
$\mathcal{Q}_{29}$	?x,?y ← ?x int+/(occ/-occ)+/(hKw/-hKw)+ ?y						×
$\mathcal{Q}_{30}$	?x ← ?x (enc/-enc | occ/-occ)+ C		×				
$\mathcal{Q}_{31}$	?x,?y ← ?x int+/(occ/-occ)+ ?y						×
$\mathcal{Q}_{32}$	?x,?y ← ?x int+/(enc/-enc)+ ?y						×
$\mathcal{Q}_{33}$	?x,?y ← ?x int/(enc/-enc)+ ?y					×	
$\mathcal{Q}_{34}$	?x,?y ← ?x -hKw/int/ref/(auth/-auth)+ ?y					×	
$\mathcal{Q}_{35}$	?x,?y ← ?x (enc/-enc)+/hKw ?y				×		
$\mathcal{Q}_{36}$	?x ← ?x (enc/-enc)+ C		×				
$\mathcal{Q}_{37}$	?x,?y,?z,?t ← ?x (enc/-enc)+ ?y, ?x int+ ?z, ?x ref ?t					×	×
$\mathcal{Q}_{38}$	?x,?y ← ?x (int|(enc/-enc))+ ?y, C (occ/-occ)+ ?y			×			×
$\mathcal{Q}_{39}$	?x ← ?x int+/ref ?y, C (auth/-auth)+ ?y			×	×		
$\mathcal{Q}_{40}$	?x ← ?x int+/ref ?y, C -pub/(auth/-auth)+ ?y			×	×	×	
$\mathcal{Q}_{41}$	?x ← C -pub/(auth/-auth)+ ?x			×		×	
$\mathcal{Q}_{42}$	?x,?y ← ?x -occ/int+/occ ?y				×	×	
$\mathcal{Q}_{43}$	?x,?y ← ?x (-ref/ref)+ ?y	×					
$\mathcal{Q}_{44}$	?x,?y ← ?x int/ref/(-ref/ref)+ ?y					×	
$\mathcal{Q}_{45}$	?x ← C (ref/-ref)+ ?x			×			
$\mathcal{Q}_{46}$	?x,?y ← ?x (-ref/ref)+/(auth|pub) ?y				×		
$\mathcal{Q}_{47}$	?x,?y ← ?x int/(occ/-occ)+ ?y					×	
$\mathcal{Q}_{48}$	?x ← C int/(enc/-enc|occ/-occ)+ ?x			×		×	
$\mathcal{Q}_{49}$	?x ← C (enc/-enc)+ ?x			×			
$\mathcal{Q}_{50}$	?x,?y ← ?x -hKw/(occ/-occ)+ ?y					×	

Figure 3.4: Uniprot queries[9].

[9] "int" stands for "interacts", "enc" for "encodes", "occ" for "occurs", "hKw" for "hasKeyword", "ref" for "reference", "auth" for "authoredBy", and "pub" for "publishes".

3.4.5 Results

We report on experimental results and analyse them. We measure the time spent in evaluating queries by the different systems, in seconds. For each set of experiments, we define a timeout value. Whenever the time spent in evaluating a query reaches this timeout value, we consider that the query evaluation did not terminate within reasonable time. On charts, the timeout value corresponds to the maximum value on the y-axis. Some systems crashed in some query evaluations. In charts, this is denoted by the presence of a red cross on a time bar. The other cases correspond to query evaluations where the system answered correctly.

Dist-μ-RA recursive plans evaluation

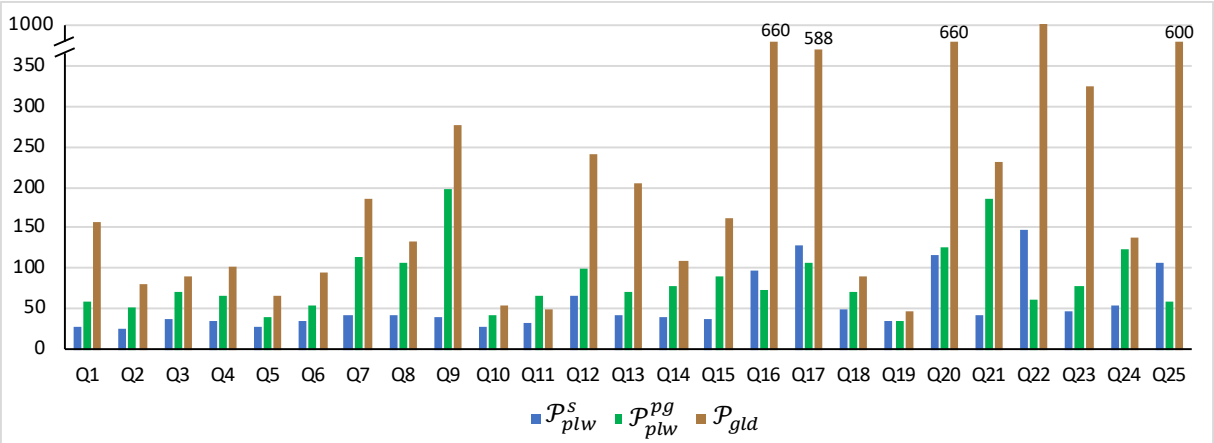


Figure 3.5: Running times of $\mathcal{P}_{\texttt{plw}}$ and $\mathcal{P}_{\texttt{gld}}$ plans on Yago.

Fig. 3.5 presents the time spent in evaluating each of the Dist-μ-RA plans (Sec. 3.3) for UCRPQs on the Yago dataset. We observe that the $\mathcal{P}_{\texttt{plw}}$ plans are faster than $\mathcal{P}_{\texttt{gld}}$. This illustrates the interest of the communication cost reduction performed by $\mathcal{P}_{\texttt{plw}}$. As explained in Sec. 3.3, $\mathcal{P}_{\texttt{gld}}$ requires communications between the workers at each step of the recursion while $\mathcal{P}_{\texttt{plw}}$ does not. Furthermore, we observe that $\mathcal{P}^{\texttt{s}}_{\texttt{plw}}$ performs better on most cases when compared to $\mathcal{P}^{\texttt{pg}}_{\texttt{plw}}$. This is due to the cost of data marshalling and exchange between PostgreSQL and Spark. However, when the size of the datasets in the variable part of the fixpoints is large, $\mathcal{P}^{\texttt{pg}}_{\texttt{plw}}$ becomes faster (e.g. queries $\mathcal{Q}_{22}$ and $\mathcal{Q}_{25}$). In that case, the performance gains are due to local optimizations of this variable part performed by PostgreSQL.

UCRPQs on Yago: comparison with other systems

In Fig. 3.6 we present the performance results of Dist-μ-RA, BigDatalog and GraphX on the queries $\mathcal{Q}_1 - \mathcal{Q}_{25}$ on the Yago dataset.

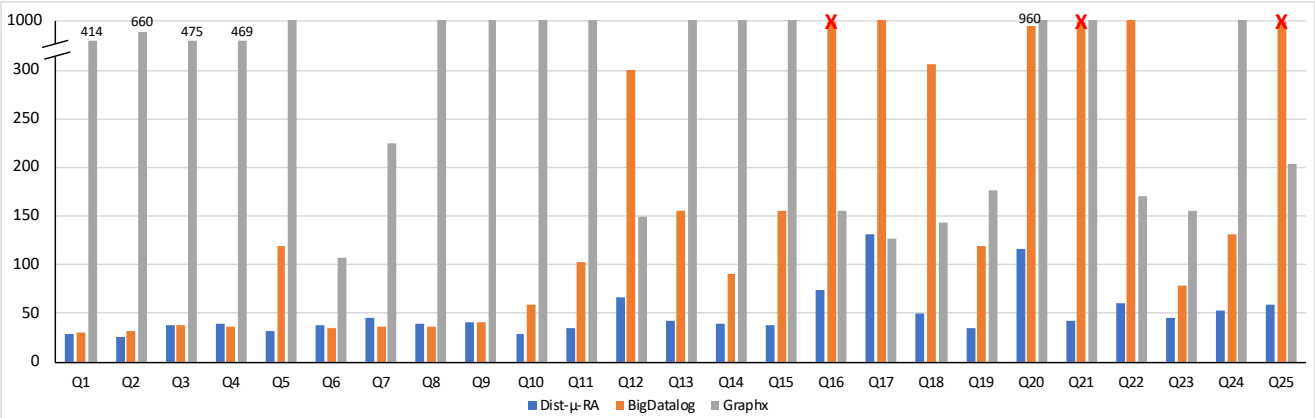


Figure 3.6: Running times on Yago. A timeout is set at 1,000 s.

First, these results show that Dist-μ-RA is much faster than GraphX overall. We believe that this lack of performance is due to the fact that, in the GraphX Pregel model, each node has to keep track of its ancestors that satisfy a given regular path query (or a part of it) and transmit this information to their successors in order to get the pairs of nodes satisfying the whole query. So while GraphX has been proven to be efficient for a number of graph algorithms [Gonzalez et al., 2014], it can be less suitable for this kind of queries. The only case where GraphX matches the performance of Dist-μ-RA is for $\mathcal{Q}_{17}$ where filtering is performed at the beginning of the query (as mentioned in Sec. 3.4.3).

Second, these results show that Dist-μ-RA provides much faster performance than Bigdatalog for all the classes $\mathcal{C}_2$-$\mathcal{C}_6$, and comparable performance for class $\mathcal{C}_1$.

One explanation for the difference in performance of $\mathcal{Q}_5$ of class $\mathcal{C}_2$ is that it requires reversing a fixpoint term first before pushing the filter "Kevin Bacon". This fixpoint reversal is not supported by Datalog's Magic Sets optimization technique (see Sec. 3.5 for more details). Another example is $\mathcal{Q}_{24}$ of class $\mathcal{C}_6$ where Dist-μ-RA merges fixpoint terms (which BigDatalog is unable to do). Overall, the optimizations in Dist-μ-RA are more effective. We noticed that this is particularly the case when the size of intermediate results is large ($\mathcal{Q}_5$ and $\mathcal{Q}_{10} - \mathcal{Q}_{25}$) .

Concatenated closures

We now evaluate concatenated closure queries (which belong to $\mathcal{C}_6$) on the graph obtained from rnd_100k_0.001. The graph edges are randomly labeled from a set of 10 different labels. Results are shown in Fig. 3.7. Dist-μ-RA is faster on all queries. The time difference between Dist-μ-RA and BigDatalog for a query with n concatenations ($a_1+/.../a_n+$) becomes larger when n increases. BigDatalog fails for queries where $n \geq 5$ and GraphX crashes on all queries. The plans that are selected in Dist-μ-RA for the execution of these queries apply a mixture of the rewritings that "push joins" and "merge fixpoints" (see Sec. 3.2). These results

also indicate that optimizations introduced by these rewritings provide significant performance gains for class $\mathcal{C}_6$.

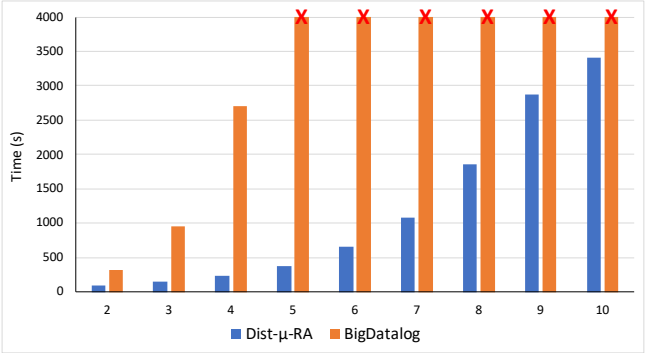


Figure 3.7: Evaluation times for concatenated closure queries.

Non regular queries

The execution times for these queries of class $\mathcal{C}_7$ are given in Fig. 3.8. On the basic SG and $a^n b^n$ queries, Dist-μ-RA and BigDatalog have comparable execution times. Dist-μ-RA is faster on Filtered SG and Joined SG queries.

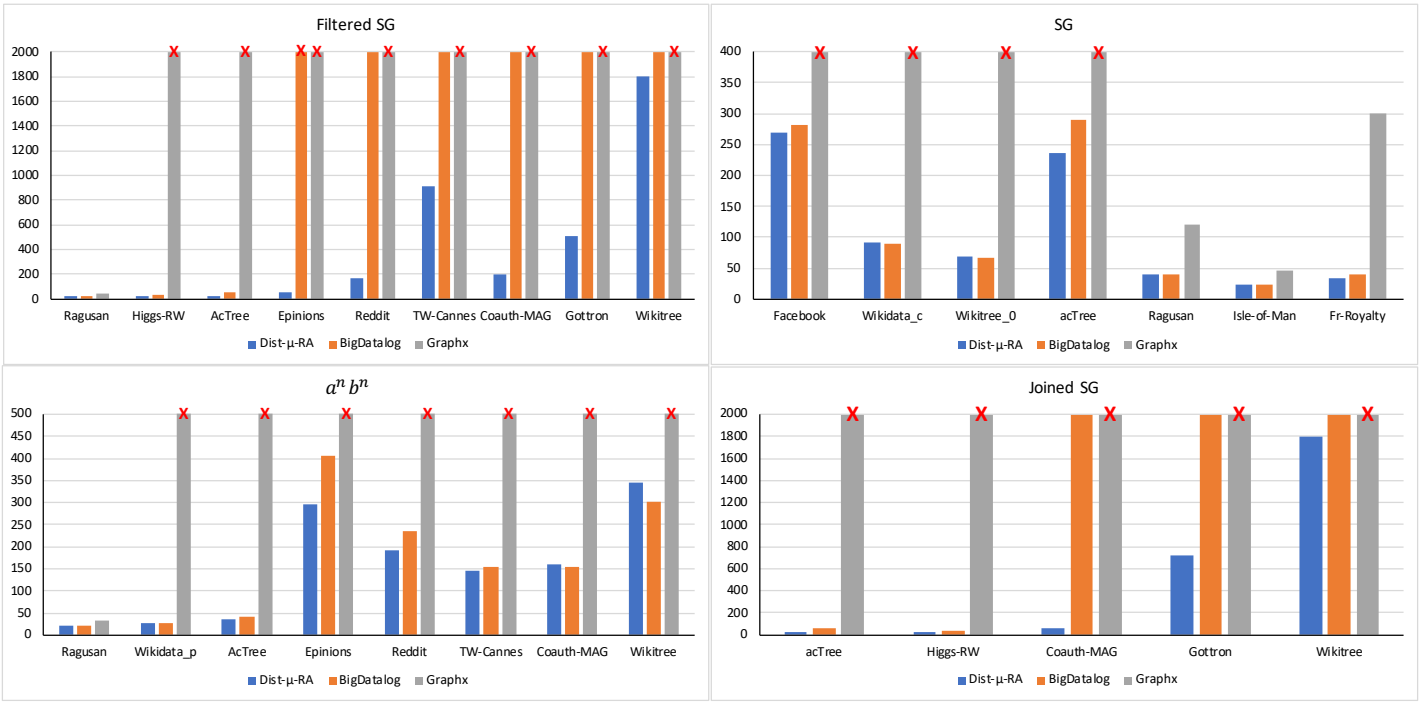


Figure 3.8: μ-RA queries running times. A timeout is set to 2000s.

The comparison between Myria and Dist-μ-RA is limited to the SG query for the reasons explained in Section 3.4.3. Dist-μ-RA is much faster than Myria for all the cases that Myria can handle. Furthermore, it can be observed in Fig. 3.9

that the difference in performance between the two systems increases with the dataset size.

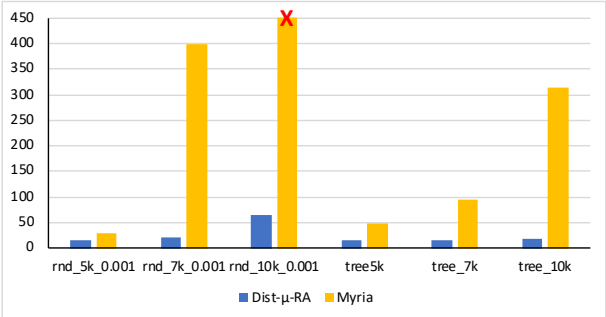


Figure 3.9: Comparison with Myria on SG.

UCRPQs on Uniprot

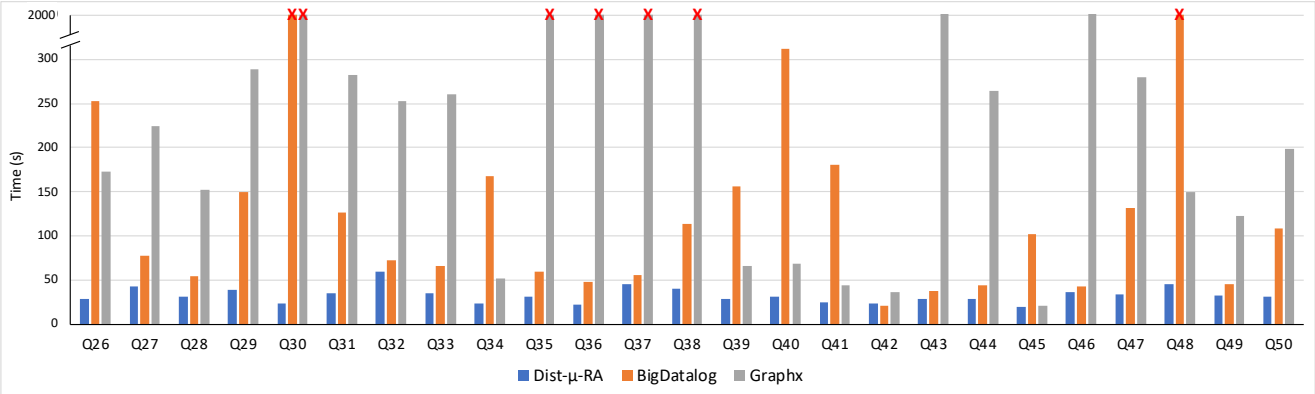


Figure 3.10: Running times on uniprot_1M. A timeout is set to 2,000 s.

The results reported in Fig 3.10 show that Dist-μ-RA is the only system that answers all of the queries. Furthermore, Dist-μ-RA is faster on all queries belonging to $\mathcal{C}_{2-6}$ (except $\mathcal{Q}_{42}$ where the size of the transitive closure is small).

The comparison with Myria is shown in Fig 3.11. Dist-μ-RA provides comparable or better performance for all queries except $\mathcal{Q}_{42}$. The latter case is due to the time spent in initializing the Spark context (around 10s), a cost always included in the measured Dist-μ-RA execution times. Myria crashes for several cases.

Further scalability tests on Uniprot We report on further scalability tests where Dist-μ-RA and BigDatalog execution times are compared for each Uniprot query on generated uniprot_n graphs with varying sizes of 1M, 5M and 10M edges. Results are shown in Fig. 3.12. Results indicate that BigDatalog fails in

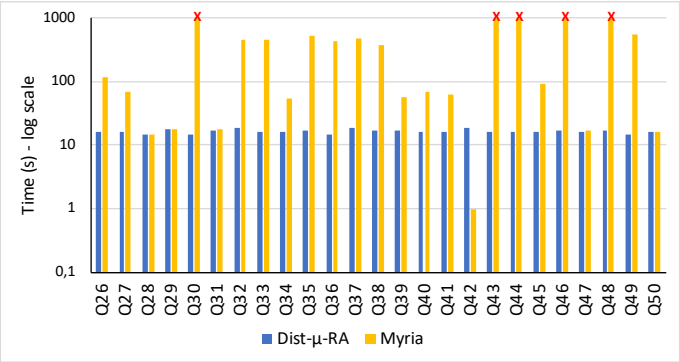


Figure 3.11: Myria and Dist-μ-RA times on uniprot_100k.

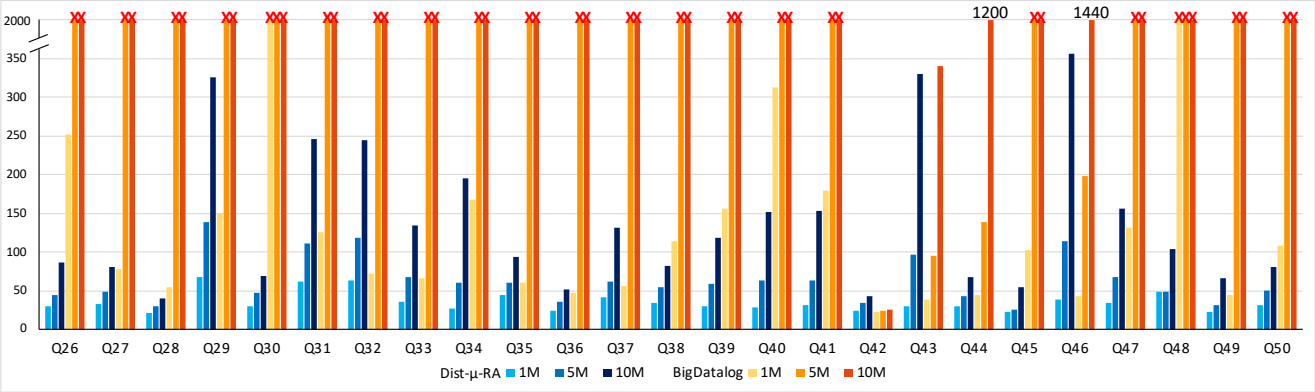


Figure 3.12: Dist-μ-RA and BigDatalog running times on Uniprot graphs of different sizes

44 cases out of 75 query evaluations. Dist-μ-RA answers all of them and scales better.

Notice that, for comprehensive testing, queries and graph sizes have been selected so as to cover a wide range of result sizes. $\mathcal{Q}_{40}$ is one of the queries with the smallest result size (14K records for uniprot_10M) and $\mathcal{Q}_{46}$ is one with the largest (around 1.5B records for uniprot_10M, which is 150 times the size of the graph).

3.4.6 Summary

Overall, for all query classes, Dist-μ-RA is significantly more efficient compared to GraphX and Myria. For query classes $\mathcal{C}_{2-6}$ and some queries in $\mathcal{C}_7$, Dist-μ-RA is more efficient than BigDatalog, especially for large intermediate query result sizes. For query class $\mathcal{C}_1$ and some queries in $\mathcal{C}_7$, Dist-μ-RA and BigDatalog have a comparable performance. Our empirical findings tend to indicate that for these cases the various optimizations techniques of Dist-μ-RA and Bigdatalog have limited impact.

3.5 Specific comparison with other distributed systems

In graph distributed systems based on Pregel (Sec.1.3.1) like Graphx, selections can be pushed in one direction only. For instance, if the program traverses the regular query from left to right, the execution of the program naturally computes the filters and edge selections occuring before a recursion first, thereby pushing these operations in the recursion. Selections which occur after the recursion cannot be pushed. In μ-RA, plans that push selections at either sides of a recursion are considered, and a cost model is used to pick the best plan. Additionally, communications between workers of the cluster happen in every superstep of the recursion, which is avoided by the $\mathcal{P}_{\texttt{plw}}$ plan in Dist-μ-RA.

Distributed systems based on Datalog (Sec. 2.4.3) suffer from the Datalog limitations for optimizing recursion presented in Sec. 2.4.2. In addition, SociaLite and Myria do not provide an optimization that avoids communication between workers at every step of a recursion. BigDatalog uses the Datalog GPS technique [Seib and Lausen, 1991] that analyses Datalog rules to identify decomposable Datalog programs and determine how to distribute data and computations. These ideas are tied to Datalog and are not applicable to the relational algebra. The present work proposes a new method specifically designed for recursive relational algebraic terms. It uses the μ-RA filter pushing technique to automatically repartition data.

3.6 Conclusion

We propose a new approach for the evaluation of recursive algebraic terms in a distributed manner. It relies on a technique capable of generating independent parallel loops on the worker nodes in a cluster of machines instead of executing a global loop on the driver node. The advantage of the parallel local loops is a minimization of the amount of data shuffled between worker nodes. This reduces communication costs and significantly improves overall query evaluation time. We applied this approach to recursive graph queries on real and synthetic datasets. Experimental results show that the proposed approach is more efficient than the state-of-the-art.

Chapter 4

Recursive monoid algebra: μ-monoids

4.1 Introduction

Ideas from functional programming play a major role in the construction of big data analytics applications. For instance they directly inspired Google's Map/Reduce [Dean and Ghemawat, 2004]. Big data frameworks (such as Spark [Zaharia et al., 2016] and Flink [Carbone et al., 2015]) further built on these ideas and became prevalent platforms for the development of large-scale data intensive applications. The core idea of these frameworks is to provide intuitive functional programming primitives for processing immutable distributed collections of data.

Writing efficient applications with these frameworks is nevertheless not trivial. Let us consider for instance the problem of finding the shortest paths in a large scale graph. We could write the Spark/Scala program in Fig 4.1 to solve it. The `shortestPaths()` function takes as input a graph `R` of weighted edges `(src, dst, weight)` and returns the shortest paths between each pair of nodes in the graph. The loop (in lines 6 to 14 of Fig 4.1) computes all the paths in the graph and their lengths; to get new paths, edges from the graph get appended to the paths found in the previous iteration using the join operation. Then reduceByKey operation is used to keep the shortest paths. Spark performs the join and distinct operations by transferring the datasets (arguments of the operations) across the workers so as to ensure that records having the same key are in the same partition for join, and that no record is repeated across the cluster for distinct. Hence, for optimizing such programs, the programmer needs to take this data exchange into account as well as other factors like the amount of data processed by each worker and its memory capacity, the network overhead incurred by shuffles, etc. One optimization that can be done to reduce data exchange in this program is to assign each worker a part of the graph and make it compute the paths in the graph that start from its own part. This optimization leads to the following program (Fig 4.2.) which is not straightforward to write, less readable, and requires the programmer to give his own local version of dataset operators (such as join) that are going to be used to perform the local computations on each worker.

```
def shortestPaths(R:RDD[(Int,Int,Int)]) = {
  var ret = R
  var X: RDD[(Int, Int, Int)] = R
  var new_cnt = ret.count()
  var cnt = new_cnt
  do {
    cnt = new_cnt
    X = X.map({case (x,y,l1) => (y,(x,l1)) })
      .join(R.map({ case (z,t,l2) => (z,(t,l2)) }))
      .map({case (_,((x,l1),(t,l2))) => (x,t,l1+l2) })
      .subtract(ret)
    ret = ret.union(X).distinct()
    new_cnt = ret.count()
  } while (new_cnt > cnt)
  ret.map({case (x,y,l) => ((x,y),l)}).reduceByKey(min)
}
```

Figure 4.1: Shortest paths program.

```
def shortestPaths(R:RDD[(Int,Int,Int)]) = {
  val dictR = LocalOps.to_dict(((x:(Int,Int,Int)) => x._1),
    (x:(Int,Int,Int)) => x, sc.broadcast(R.collect()).value)
  var r = R.mapPartitions(part => {
    var ret = part.toList
    var X = ret
    var cnt = ret.size
    var new_cnt = cnt
    do {
      count = new_count
      X = LocalOps.join(LocalOps.to_dict(((x:(Int,Int,Int)) => x._2),
          (x:(Int,Int,Int)) => x, X), dictR)
        .map({case (k, ((x,y,l), (a,b,m))) => (x,b,l+m)}) diff ret
      ret = (ret ++ X).distinct
      new_count = ret.size
    } while (new_cnt > cnt)
    ret.toIterator
  })
  r.distinct().map({case (x,y,l) => ((x,y),l)}).reduceByKey(min)
}
```

Figure 4.2: Shortest paths program with less data exchange.

Another possible optimization is to put the `reduceByKey` operation inside the loop to keep only the shortest paths at each iteration because each subpath of a shortest path is necessarily a shortest path. More generally, finding such program rewritings can be hard. First, it requires guessing which program parts affect performance the most and could potentially be rewritten more efficiently. Second, assessing that the rewriting performs better can hardly be determined without experiments. During such experiments, the programmer might rewrite the program possibly several times, because he has limited clues of which combination of rewritings actually improves performance.

In this chapter, we explore the foundations for the automatic transformation and optimization of Spark programs. Algebraic foundations in particular are an active research topic [Alexandrov et al., 2019, Fegaras, 2017]. The purpose of the algebraic formalism is to represent a program in terms of algebraic operators that can be analysed and transformed so as to produce a program that executes faster. Transformation-based optimizations are done through rewrite rules that transform an algebraic expression to an equivalent, yet more efficient, one. In the context of big data applications, considered algebras must be able to capture distributed programs on big data platforms and provide the appropriate primitives to allow for their optimization. One example of optimizations is to push computations as close as possible to where data reside.

When programming with big data frameworks, data is usually split into partitions and both data partitions and computations are distributed to several machines. These partitions are processed in parallel and intermediate results coming from different machines are combined, so that a unique final result is obtained, regardless of how data was split initially. This imposes a few constraints on computations that combine intermediate results. Typically, functions used as aggregators must be associative. For this reason, we consider that the monoid algebra is a suitable algebraic foundation for taking this constraint into account at its core. It provides operations that are monoid homomorphisms, which means that they can be broken down to the application of an associative operator. This associativity implies that parts of the computation can actually be performed in parallel and combined to get the final result.

A significant class of big data programs are iterative or recursive in nature (PageRank, k-means, shortest-path, reachability, etc.). Iterations and recursions can be implemented with loops. Depending on the nature of the computations performed inside a loop, the loop might be evaluated in a distributed manner or not. Furthermore, certain loops that can be distributed might be evaluated in several ways (global loop on the driver[1], parallel loops on the workers, or a nested combination of the latter). The way loops are evaluated in a distributed setting often has a great impact on the overall program execution cost. Obviously, the task of identifying which loops of an entire program can be reorganized into more efficient distributed variants is challenging. This often constitutes a major obstacle for automatic program optimization. In the algebraic formalism, having a recursion operator makes it possible to express recursion while abstracting away

[1] In Big Data frameworks such as Spark, the *driver* is the process that creates tasks and sends them to be executed in parallel by *worker* nodes. See Sec. 1.3.1.

from how it is executed. The execution plan is then decided after analysing the program.

The goal of this work is to introduce a gain in automation of distributed program transformation towards more efficient variants. We focus especially on recursive programs (that compute a fixpoint). For this purpose, we propose an algebra capable of capturing the basic operations of distributed computations that occur in big data frameworks, and that makes it possible to express rewriting rules that rearrange the basic operations so as to optimize the program. We build on the monoid algebra introduced in [Fegaras and Noor, 2018, Fegaras, 2017] that we extend with an operator for expressing recursion. This monoid algebra is able to model a subset of a programming language $\mathcal{L}$ (for instance Scala), that expresses computations on distributed platforms (for instance Spark).

The novelty is twofold:

1. An extension of the monoid algebra with a fixpoint operator. This enables the expression of iteration in a more functional way than an imperative loop and makes it possible to define new rewriting rules;
2. New optimization rules for terms using this fixpoint operator:
   - We show that under reasonable conditions, this fixpoint can be considered as a monoid homomorphism, and can thus be evaluated by parallel loops with one final merge rather than by a global loop requiring network overhead after each iteration;
   - We also present new rewriting rules with criteria to push filters through a recursive term, for filtering inside a fixpoint before a join, and for pushing aggregations into recursive terms;
   - Finally, we present experimental evidence that these new rules generate significantly more efficient programs.

4.2 The μ-monoids Algebra

In this section, we describe a core calculus, which we call μ-monoids, intended to model a subset of a programming language $\mathcal{L}$ (e.g. Scala[2]) that is used for computations on a big data framework (through an API provided by the framework). μ-monoids aims at being as general as possible, while focusing on formalizing computations subject to optimization. Dataset manipulations are captured as algebraic operations, and specific operations on elements of those datasets are captured as functional expressions that are passed as arguments to some of the algebraic operations. In μ-monoids, we formalize *some* of those functional constructs, specifically the ones that we need to analyse in the algebraic expressions. For example, some optimization rules need to analyse the pattern and body of flatmap expressions in order to check whether the optimization can take place.

[2]Major Bigdata frameworks like Spark and Flink provide a Scala API and are implemented in Scala which makes Scala a suitable language for our work. Scala also provides reflection which allows generic Scala constructs to be part of the algebra as we will explain later.

Making explicit only the shapes that are interesting for the analysis enables to abstract from the specific programming language $\mathcal{L}$ that we optimize. This way, constructs of $\mathcal{L}$ other than those which we model explicitly are represented as *constants* c, as they are going to be left to $\mathcal{L}$'s compiler to typecheck and evaluate. We only assume that every constant c has a type $type(c)$ which is either a basic type or a function type, and that, when its type is $t_1 \to t_2$, it can be applied to any argument of type t_1 to yield results of type t_2.

We first describe the data model we consider, then in Sec. 4.2.2 we introduce the syntax of our core calculus. We then proceed to give a denotational semantics for our specific constructs in Sec. 4.2.4 and discuss evaluation of expressions in Sec. 4.2.6.

4.2.1 Data model: distributed collections of data

Collection monoids

We are interested in programs which work on distributed datasets of an homogenous type. Such a dataset consists in a number of records, which are all values of the same type, and we assume a cluster of networked machines where each machine stores some of the records.

Different abstraction levels are possible for such a distributed dataset. At the programming level, we usually want to abstract away from the partitioning, i. e. we consider two states of the storage as representing the same data if they contain the same records, regardless of the number of machines and of which machine holds which records. That way, the program is reasonably independent from the structure of the cluster it will be run on. We may or may not want to abstract away from the order in which the records are stored, and we may or may not want to abstract away from the number of times the same record appears. Depending of the abstraction level, we thus can see the dataset as a list, a bag, or a set of records. We regroup finite lists, finite bags and finite sets under the generic term of *collections*.

Notation. Given a data type t, and for *Coll* a sort of collection, i. e. one of $\mathsf{List}, \mathsf{Bag}$ or FSet, we write $Coll[t]$ for the set of collections of the sort *Coll* containing values of type t.

As noticed by Fegaras [Fegaras, 2017], these sorts of collections are particularly useful for representing distributed data because they each have the algebraic structure of a monoid, where the neutral element is the empty collection and the associative operator is respectively list union (i. e. concatenation) $+\!\!+$, bag union $\uplus$ and set union $\cup$. Associativity means that the whole collection can be seen as the union of the subcollections stored on the different machines without specifying an order in which to apply the union operator.

The three sorts of *collection monoids*, as Fegaras terms them, can be related with equivalence relations, reflecting the fact that they represent different abstraction levels for the same data: let $\sim_{\text{comm}}$ be the congruence on $(\mathsf{List}[t], +\!\!+, [\,])$ generated by commutativity, i. e. the smallest equivalence relation on $\mathsf{List}[t]$ such that:

- $\forall a, b \quad a +\!\!+ b \sim_{\text{comm}} b +\!\!+ a$ (commutativity)
- $\forall a, b, a', b' \quad a \sim_{\text{comm}} a' \wedge b \sim_{\text{comm}} b' \Rightarrow a +\!\!+ b \sim_{\text{comm}} a' +\!\!+ b'$ (compatibility with $+\!\!+$)

This relation relates all lists containing exactly the same elements, with the same multiplicity, in any order. So a *bag* can be seen as an equivalence class of lists for $\sim_{\text{comm}}$, meaning that $(\mathsf{Bag}[t], \uplus, \{\!\{\}\!\})$ is actually the quotient monoid $(\mathsf{List}[t], +\!\!+, [\,])/\sim_{\text{comm}}$. Similarly, let $\sim_{\text{idem}}$ be the congruence on bags generated by idempotence ($a \uplus a \sim_{\text{idem}} a$): it relates all bags containing the same elements, regardless of their multiplicity, and we have that $(\mathsf{FSet}[t], \cup, \emptyset)$ is the quotient monoid $(\mathsf{Bag}[t], \uplus, \{\!\{\}\!\})/\sim_{\text{idem}}$.

In this work, we choose bags as the default base abstraction level, since there is no canonical ordering of the machines in the cluster; but this can be adapted to work with lists. So from now on, we consider that a dataset is a distributed bag. We now define the formal syntax of our data model before developing further how we can sometimes work up to equivalence relations if, e. g., we are in fact interested in sets and not bags.

Values and types of μ-monoids

In order to enable algebraic datatypes, we assume an infinite set of *constructors* C which can be applied to any number of values. We assume this set contains the special constructors True, False and Tuple for which we will define some syntactic sugar.

The syntax of considered data values is defined as follows:

v	::=	c	constant
	|	$C(v_1, v_2, ..., v_n)$	n-ary constructor
	|	$\{\!\{v_1, ..., v_n\}\!\}$	bag

As mentioned previously (Sec. 4.2), a constant c can be any value from the language $\mathcal{L}$ (in particular any function) that is not explicitly defined in our syntax.

We define the following syntax for types:

t_l	::=		local type	t	::=	type	
		$\mathbb{B}$	basic type		|	t_l	
	|	$C_1[t_l, ..., t_l] \,\|\|\, \cdots \,\|\|\, C_n[t_l, ..., t_l]$	sum type		|	$\mathsf{Bag}_d[t_l]$	distributed bag type
	|	$\mathsf{Bag}_l[t_l]$	local bag type		|	$t \to t$	function type

where $\mathbb{B}$ represents any arbitrary basic type (i.e., considered as a constant atomic type in our formalism).

In sum types, all constructors have to be different and their order is irrelevant. They represent values which can belong to any of the case types $C_1[t_l, ..., t_l] ... C_n[t_l, ..., t_l]$ and can be deconstructed by pattern-matching.

We also define product types $t_1 \times \cdots \times t_n$ as syntactic sugar for $\mathsf{Tuple}[t_1, ..., t_n]$, i. e. a particular case of constructor type.

For a given type t, we denote by $\mathsf{Bag}_l[t]$ the type of a local bag and by $\mathsf{Bag}_d[t]$ the type of a distributed bag of values of type t. Notice that we can

have distributed bags of any data type t including local bags, which allows us to have nested collections. We allow data distribution only at the top level though (distributed bags cannot be nested).

An important feature of Fegaras' monoid algebra, and of μ-monoids, is that all algebraic operations are defined in a way which is agnostic to distribution. So, although we introduce the distinction between $\mathsf{Bag}_l[t]$ and $\mathsf{Bag}_d[t]$ in order to prevent nesting distributed bags, we will use the notation $\mathsf{Bag}[t]$ to represent a bag which may or may not be distributed when both are possible and it does not affect the semantics.

Equivalence relations and aggregation functions

It is quite common in practice that a programmer is only interested in the set of values of a dataset, not in potential duplicates the bag representing the storage may contain. So this programmer will work with bags up to $\sim_{\text{idem}}$ (see Sec. 4.2.1). We can notice that each equivalence class of bags has a canonical representant: the bag where each element appears only once. Let $\mathsf{distinct} : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ be the function which removes duplicates, returning this canonical representant. This function is useful, but costly to compute in a distributed context, since duplicates can occur across different machines and eliminating them thus involves a lot of communication over the network. Therefore, it should not be used all the time but only when necessary: bags with duplicates can be used in intermediary computation steps, where we tolerate redundant information temporarily.

Sometimes, the programmer is not even interested in the whole set of values, but only in more synthetic information about the dataset. For example, in the shortest path problem: if we have a dataset containing paths together with their length and this dataset contains two paths from a to b with different lengths, then the longer path is irrelevant and can be considered redundant even though it is not the same value as the other one. It is useful to also think of this situation in terms of an equivalence relation: two bags are equivalent for this purpose iff they contain exactly the same *shortest* paths. Then the canonical representant of an equivalence class is the bag with no duplicates which does not contain any non-shortest path. We can also see that the function δ which removes non-shortest paths (and duplicates) from a dataset has features analogous to $\mathsf{distinct}$, as we will detail below. We regroup such functions under the term *aggregation functions*.

Definition 6 (aggregation function)**.** *We call* aggregation function *any function* $\delta : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ *with the following properties:*

- $\delta(\{\!\{\}\!\}) = \{\!\{\}\!\}$
- $\forall a, b \quad \delta(a \uplus b) = \delta(\delta(a) \uplus \delta(b))$

Note that these two properties also imply that δ is idempotent.

Remark that our definition excludes some functions which could be considered aggregators in a more general sense, e. g. functions computing an average.

Definition 7. *The equivalence relation associated with an aggregation function, $\sim_\delta$, is defined by:*

$$a \sim_\delta b \stackrel{def}{\Leftrightarrow} \delta(a) = \delta(b)$$

Lemma 1. *Let δ be an aggregation function, then $\sim_\delta$ is compatible with the monoid operation $\uplus$, i. e. we have:*

$$\forall a, b, a', b' \quad a \sim_\delta a' \wedge b \sim_\delta b' \Rightarrow a \uplus b \sim_\delta a' \uplus b'$$

Proof. We have $\delta(a \uplus b) = \delta(\delta(a) \uplus \delta(b)) = \delta(\delta(a') \uplus \delta(b')) = \delta(a' \uplus b')$. □

Definition 8. *Let δ be an aggregation function, we define the binary operation $\otimes_\delta$ on bags as follows:*

$$a \otimes_\delta b \stackrel{def}{=} \delta(a \uplus b)$$

Lemma 2. *Let $\delta(\mathsf{Bag}[t])$ be the image of δ. Then $(\delta(\mathsf{Bag}[t]), \otimes_\delta, \{\!\{\}\!\})$ is a monoid, noted M_δ, isomorphic to the quotient monoid $\mathsf{Bag}[t]/\sim_\delta$, and δ is a monoid homomorphism: $\mathsf{Bag}[t] \to M_\delta$.*

Proof. Since δ is idempotent, we can consider $\delta(a)$ the canonical representant of the equivalence class of a; then we have an isomorphism between the equivalence classes (the monoid $\mathsf{Bag}[t]/\sim_\delta$) and their canonical representants (the monoid M_δ). □

Example 4. *The function $\mathsf{distinct} : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ which removes duplicates from a bag is an aggregation function; $\sim_{\mathsf{distinct}}$ is the relation $\sim_{idem}$; $\otimes_{\mathsf{distinct}}$ is distinct union of bags $\cup$; and M_{distinct} is the monoid of bags with no duplicates, isomorphic to $\mathsf{FSet}[t]$.*

Finally, in order to work up to equivalence relations, we need the notion of compatibility between an homomorphism φ from bags to bags and an aggregation function δ:

Lemma 3. *Let $\varphi : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ be a monoid homomorphism and $\delta : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ an aggregation function. The three following properties are equivalent:*

1. $\forall a, b \quad a \sim_\delta b \Rightarrow \varphi(a) \sim_\delta \varphi(b)$
2. $\forall a, b \quad \varphi(a \otimes_\delta b) \sim_\delta \varphi(a) \otimes_\delta \varphi(b)$
3. $\delta \circ \varphi \circ \delta = \delta \circ \varphi$

Proof. Assume (1) is true. Let a and b be any bags. We have $\varphi(a) \otimes_\delta \varphi(b) = \delta(\varphi(a) \uplus \varphi(b)) = \delta(\varphi(a \uplus b))$ (because φ is a homomorphism). We also have $a \uplus b \sim_\delta \delta(a \uplus b)$, by definition of $\sim_\delta$ since δ is idempotent. Therefore, using (1), $\varphi(a \uplus b) \sim_\delta \varphi(\delta(a \uplus b))$, and this last term is $\varphi(a \otimes_\delta b)$; hence (2).

Assume (2) is true. Let a be any bag, by taking for b the empty bag and using the definitions, (2) yields $\delta(\varphi(\delta(a \uplus \{\!\{\}\!\}))) = \delta(\varphi(a) \uplus \varphi(\{\!\{\}\!\}))$. Since φ is an homomorphism we have $\varphi(\{\!\{\}\!\}) = \{\!\{\}\!\}$, thus $\delta(\varphi(\delta(a))) = \delta(\varphi(a))$; hence (3).

Assume (3) is true. Let a and b such that $a \sim_\delta b$. Using (3) and the definition of $\sim_\delta$, we have $\delta(\varphi(a)) = \delta(\varphi(\delta(a))) = \delta(\varphi(\delta(b))) = \delta(\varphi(b))$; hence (1). □

Definition 9. *We say that φ is* compatible *with δ if any of these properties is true. Note that (2) can also be formulated as: $\delta \circ \varphi$ is a monoid homomorphism from M_δ to M_δ.*

This definition is strongly related to the premappability condition in Datalog [Zaniolo et al., 2017], as made more apparent by property (3).

Property 1. *distinct is compatible with all homomorphisms $\varphi : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$.*

Proof. In the following, we write $m \cdot X$, where X is a bag, to denote X combined m times with itself using $\uplus$.

Let A be a finite bag of values of type t. Let $a_1, ..., a_n$ be the distinct values it contains and $m_1, ..., m_n$ the number of times each one appears in the bag. Since φ is an homomorphism, we have $\varphi(A) = \biguplus_{1 \leq i \leq n} m_i \cdot \varphi(\{\!\{a_i\}\!\})$. Then, $\mathsf{distinct}(\varphi(A)) = \bigcup_{1 \leq i \leq n} \mathsf{distinct}(\varphi(\{\!\{a_i\}\!\}))$: this is independent from the m_i (since $\cup$ is idempotent). Thus, $\mathsf{distinct}(\varphi(A)) = \mathsf{distinct}(\varphi(\mathsf{distinct}(A)))$. □

4.2.2 The μ-monoids syntax

Monoid homomorphisms and μ operation

Our syntax is built on the monoid algebra proposed by Fegaras [Fegaras, 2017]. In addition to the general-purpose constructs provided by the language $\mathcal{L}$, it contains the following primitives to work on bags (distributed or not):

- $\mathsf{flmap}(f, X)$, with $f : t_1 \to \mathsf{Bag}_l[t_2]$ and $X : \mathsf{Bag}[t_1]$ is the flatmap operation: it applies f to each element of X and merges all the results into a single bag using bag union $\uplus$. This operation is a monoid homomorphism from $\mathsf{Bag}[t_1]$ to $\mathsf{Bag}[t_2]$, so that if X is distributed it can be run separately on each local subcollection without any data exchange. Note the restriction that f is not allowed to return distributed bags. It makes the flatmap operator less general than the mathematical function it represents but reflects what we have in distributed data frameworks.

- $\mathsf{reduce}(\oplus, e_\oplus, X)$, with $X : \mathsf{Bag}[t]$ and $(t, \oplus, e_\oplus)$ a commutative monoid, reduces the input dataset by combining all its elements with $\oplus$. For example: $\mathsf{reduce}(+, 0, \{\!\{1, 4, 6\}\!\}) = 11$. This operation is a monoid homomorphism from $\mathsf{Bag}[t]$ to $(t, \oplus, e_\oplus)$, so that if X is distributed it can be run separately on each local subcollection before combining all the local results once.

- $\mathsf{reduceByKey}(\oplus, X)$, with $X : \mathsf{Bag}[t_1 \times t_2]$ and $\oplus$ an associative and commutative binary operation on t_2, takes as argument a bag of elements in the form (k, v) (key-value pair) and combines all values v having the same key k into a single one using the $\oplus$ operator. For example: $\mathsf{reduceByKey}(+, \{\!\{(1,2), (1,4), (2,2), (2,1), (1,3)\}\!\}) = \{\!\{(1,9), (2,3)\}\!\}$. This operation is an aggregation function (see Def. 6), and therefore also a monoid homomorphism (Lemma 2) which can again be run separately on each local subcollection before combining the results.

- $\mathsf{join}(X, Y)$, with $X : \mathsf{Bag}[t_1 \times t_2]$ and $Y : \mathsf{Bag}[t_1 \times t_3]$, is the join-by-key operation: it takes two collections of elements of the form (k, v) and (k, w), and returns a collection of elements of the form $(k, (v, w))$, one for each pair (v, w) of values having the same key k. If a key appears n times in one input dataset and m times in the other, it appears nm times in the result. It is a monoid homomorphism from each of its arguments to $\mathsf{Bag}[t_1 \times (t_2 \times t_3)]$, so that if any of the input bags is distributed it can be run separately on each local subcollection for that one.

  Note that, algebraically, the join operation can be written with flatmaps (this is a feature of all homomorphisms from bags to bags); however, if both inputs are distributed then this is not possible in μ-monoids without violating the restriction on the functional argument of flatmap, which justifies including join as a primitive.

- $\mathsf{cogroup}(X, Y)$, with $X : \mathsf{Bag}[t_1 \times t_2]$ and $Y : \mathsf{Bag}[t_1 \times t_3]$, takes two collections of elements of the form (k, v) and (k, w) and returns a collection of elements of the form $(k, (V, W))$ where V and W are the sets of v values and w values having the same key k.

Our purpose is to extend this algebra with an operator for expressing iteration which allows effective optimisations when working with a distributed dataset (note that this does not preclude more general loops outside our algebra).

As a first idea, consider the following type of iteration. Let $\varphi : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ be a monoid homomorphism. We start with a bag R, then:

1. at each iteration, φ is executed on the result of the previous iteration;
2. the results of all iterations are accumulated into a single bag;
3. it ends when φ adds nothing to the results; then the bag of all results is returned.

Algebraically, this amounts to computing $\biguplus_{n \in \mathbb{N}} \varphi^n(R)$. The fact that φ is a monoid homomorphism implies that, if R is distributed, such a loop can be executed separately on each sub-bag, with no communication necessary, which is very good for efficiency. However, if in fact we are not interested in bags with duplicates but only in sets, i. e. if we work up to $\sim_{\text{idem}}$, it has the serious drawback that it only stops when φ returns the empty bag: this can prevent termination in cases where φ generates nothing really new but adds indefinitely more duplicates. A typical example is if we want to compute the transitive closure of a relation with cycles.

Therefore, it makes sense to periodically remove duplicates. However, it may not be necessary to remove them *globally* (which is costly as it involves network communication), as we will detail in Section 4.3.4. More generally, we can add to the loop, as a parameter, an aggregation function δ to be run at each iteration step. Our general iteration operator μ is thus:

Definition 10 (μ operator)**.** *Let $\varphi : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ be a monoid homomorphism, $\delta : \mathsf{Bag}[t] \to \mathsf{Bag}[t]$ an aggregation function, and $R : \mathsf{Bag}[t]$ a dataset. Assume φ*

and δ are compatible (Def. 9). The operation $\mu_\delta\,(R, \varphi)$ computes the following sequences:

- $R_0 = \delta(R)$
- $S_0 = R_0$
- $R_{n+1} = \delta(\varphi(R_n))$
- $S_{n+1} = S_n \otimes_\delta R_{n+1}$

until it reaches an N such that $S_{N+1} = S_N$; then it returns S_N.

Note that the first idea discussed before is the particular case where δ is the identity function. Also note that the requirement that δ and φ are compatible is automatically true when δ is either the identity function or distinct.

In the following, we consider distinct the default aggregation function and write $\mu\,(R, \varphi)$ as a shortcut for $\mu_{\mathsf{distinct}}\,(R, \varphi)$. Our idea is that programmers would usually not write μ terms with a different δ themselves, but they can be obtained through rewriting and used for optimization (Sec. 4.3.3).

Syntax

The syntax of expressions is formally defined as follows:

π	::=	$a \mid C(\pi_1, \pi_2, ..., \pi_n)$	pattern: variable, constructor pattern
e	::=	$c \mid a \mid \{\!\{e\}\!\}$	expression: constant, variable, singleton
	|	$\lambda\,\langle \pi_1 \to e_1 \mid \cdots \mid \pi_n \to e_n \rangle$	function with pattern matching
	|	$e\ e \mid C(e_1, e_2, ..., e_n)$	application, constructor expression
	|	$\mathsf{flmap}(e, e) \mid \mathsf{reduce}(e, e_e, e)e$	flatmap, reduce
	|	$\mathsf{reduceByKey}(e, e) \mid \mathsf{cogroup}(e, e) \mid \mathsf{join}(e, e)$	reduce by key, cogroup, join by key
	|	$\mu_e\,(e, e)$	fixpoint

To this, we add the following as syntactic sugar:

- $(e_1, ..., e_n)$ with no constructor is an abbreviation for: $\mathsf{Tuple}(e_1, ..., e_n)$
- if e then e_1 else e_2 is an abbreviation for: $\lambda\,\langle \mathsf{True} \to e_1 \mid \mathsf{False} \to e_2 \rangle\ e$, i. e. a particular case of pattern-matching against the two constant constructors True and False representing Boolean values.
- $\mathsf{groupby}(e)$ is an abbreviation for: $\mathsf{reduceByKey}(\uplus, \mathsf{flmap}(\lambda\ \langle (k, v) \to (k, \{\!\{v\}\!\}) \rangle, e))$
- Constants c can also represent functions (defined in the language $\mathcal{L}$). We consider operators such as the bag union operator $\uplus$ as constant functions of two arguments and use the infix notation as syntactic sugar.
- To make examples more readable, we use the let **name** $= e_1$ in e_2 syntax with the usual meaning.

Example:

$$\begin{array}{l}\mathsf{let\ appendToWords} = \lambda\ \langle X \to \\ \quad \mathsf{flmap}(\lambda\ \langle x \to \mathsf{flmap}(\lambda\ \langle c \to \mathsf{if\ contains}\, x\, c\ \mathsf{then}\ \{\!\{\}\!\}\ \mathsf{else}\ \{\!\{x + c\}\!\}\rangle, C)\rangle, X)\rangle \\ \mathsf{in}\ \mu(C, \mathsf{appendToWords})\end{array}$$

This expression computes the set of all possible words with no repeated letters that can be formed from a set of characters C. We assume that + and contains are defined in $\mathcal{L}$: + appends its second argument to the first and contains checks whether the first argument is contained in the second argument. appendToWords thus returns a new set of words from a given set of words X by appending to each of the words in X each letter in C whenever it was not already present.

The iteration operator computes the following, where we consider $C = \{\!\{a, b, c\}\!\}$ — the fixpoint is reached in 3 steps:

$$\begin{array}{ll} R_0 = C & S_0 = C \\ R_1 = \mathsf{distinct}(\varphi(C)) = \{\!\{ab, ac, ba, bc, ca, cb\}\!\} & S_1 = C \cup R_1 = \{\!\{ab, ac, ba, bc, ca, cb, a, b, c\}\!\} \\ R_2 = \{\!\{abc, acb, bac, bca, cab, cba\}\!\} & S_2 = \{\!\{abc, acb, bac, bca, cab, cba, ab, ac, ba, bc, ca, a, b, c\}\!\} \\ R_3 = \mathsf{distinct}(\varphi(R_2)) = \{\!\{\}\!\} & S_3 = S_2 \cup R_3 = S_2 \end{array}$$

4.2.3 Well-typed terms

We define typing rules for algebraic terms, in order to exclude meaningless terms. In these rules, we use *type environments* Γ which bind variables to types. An environment contains at most one binding for a given variable. We combine them in two different ways:

- $\Gamma \cup \Gamma'$ is only defined if Γ and Γ' have no variable in common, and is the union of all bindings in Γ and Γ';
- $\Gamma + \Gamma'$ is defined by taking all bindings in Γ' plus all bindings in Γ for variables not appearing in Γ'. In other words, if a variable appears in both, the binding in Γ' overrides the one in Γ.

Definition 11 (matching)**.** *We first define the environment obtained by* matching *a data type to a pattern by the following:*

$$\frac{}{match(a, t) \to a : t}$$

$$\frac{\forall i\ \ match(\pi_i, t_i) \to \Gamma_i}{match(C(\pi_1, ..., \pi_n), C[t_1, ..., t_n]) \to \Gamma_1 \cup \cdots \cup \Gamma_n}$$

If, according to these rules, there is no Γ such that $match(\pi, t) \to \Gamma$ holds, we say that pattern π is incompatible *with type t. Note that, with our conditions, a pattern containing several occurrences of the same variable is not compatible with any type and hence cannot appear in a well-typed term, as the typing rules will show.*

In order to type functions with pattern-matching, we define the following operation for combining sum types:

Definition 12. *The operation $+$ on sum types is defined recursively as follows. Let t be a sum type and C a constructor not appearing in t, then:*

$$t + (t'_1 \mid\mid \cdots \mid\mid t'_m) = (t + t'_1) + (t'_2 \mid\mid \cdots \mid\mid t'_m)$$
$$t + C[t_1, ..., t_n] = t \mid\mid C[t_1, ..., t_n]$$
$$(t \mid\mid C[t_1, ..., t_n]) + C[t'_1, ..., t'_n] = t \mid\mid C[t_1 + t'_1, ..., t_n + t'_n]$$

If t is not a sum type, we define $t + t = t$. The type $t + t'$ is not defined if $t \neq t'$ and t or t' is not a sum type, or if they have constructors in common with incompatible type parameters, i. e. type parameters which cannot themselves be combined with $+$.

Definition 13 (Subtyping)**.** *We define subtyping as follows (it is nontrivial only for sum types):*

$$t <: t' \overset{def}{\Leftrightarrow} t + t' = t'$$

Definition 14 (Well-typed terms)**.** *A term e is* well-typed *in a given environment Γ iff $\Gamma \vdash e : t$ for some type t, as judged by the relation defined in Figure 4.3. In these rules, T represents one of Bag_l or Bag_d.*

Note that these rules do not give a way to infer the parameter type of a λ expression in general; we assume some mechanism for that in the language $\mathcal{L}$.

Additional restrictions

In addition to the constraints imposed by our type system, some operations require their operands to fulfill certain criteria in order to be well-defined:

- $\mathsf{reduce}(f, z, A)$ and $\mathsf{reduceByKey}(f, A)$: f is associative and commutative, and z is a neutral element for f.
- $\mu_\delta\,(R, \varphi)$: φ is a monoid homomorphism, δ is an aggregation function, and they are compatible.

The user needs to provide terms that satisfy these criteria since they cannot be verified statically in general. However, regarding the homomorphism criterion for φ, even though we cannot check statically whether an arbitrary function is a monoid homomorphism, we can identify a subset of functions that can be statically checked. It is the set of terms φ of the form $\lambda\,\langle X \to \textsc{h}(X)\rangle$ where $\textsc{h}(X)$ is defined as follows:

$$\begin{array}{lll} \textsc{h}(X) & ::= & \\ & X & \\ & \mid\ \mathsf{flmap}(f, \textsc{h}(X)) & X \text{ does not appear in } f \\ & \mid\ \mathsf{join}(\textsc{h}(X), A) & X \text{ does not appear in } A \\ & \mid\ \mathsf{join}(A, \textsc{h}(X)) & X \text{ does not appear in } A \end{array}$$

This set of terms is in fact quite general: indeed, we know from algebra that homomorphisms from $\mathsf{Bag}[t]$ to $\mathsf{Bag}[t]$ are in one-to-one correspondance with functions from t to $\mathsf{Bag}[t]$, *via* the general flatmap operation[3]. In our case,

[3]This is due to the universal property of $\mathsf{Bag}[t]$, which is the free commutative monoid on t.

$$\frac{\Gamma \vdash e_1 : t \to \mathsf{Bag}_l[t'] \qquad \Gamma \vdash e_2 : T[t]}{\Gamma \vdash \mathsf{flmap}(e_1, e_2) : T[t']}$$

$$\frac{\Gamma \vdash e_1 : t \to t \to t \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : T[t]}{\Gamma \vdash \mathsf{reduce}(e_1, e_2, e_3) : t}$$

$$\frac{\Gamma \vdash e_1 : t' \to t' \to t' \qquad \Gamma \vdash e_2 : T[t \times t']}{\Gamma \vdash \mathsf{reduceByKey}(e_1, e_2) : T[t \times t']}$$

$$\frac{\Gamma \vdash e_1 : T_1[t \times t_1] \qquad \Gamma \vdash e_2 : T_2[t \times t_2] \qquad T_3 = (\mathsf{if}\ T_1 = T_2\ \mathsf{then}\ T_1\ \mathsf{else}\ \mathsf{Bag}_d)}{\Gamma \vdash \mathsf{cogroup}(e_1, e_2) : T_3[t \times (\mathsf{Bag}_l[t_1] \times \mathsf{Bag}_l[t_2])]}$$

$$\frac{\Gamma \vdash e_1 : T_1[t \times t_1] \qquad \Gamma \vdash e_2 : T_2[t \times t_2] \qquad T_3 = (\mathsf{if}\ T_1 = T_2\ \mathsf{then}\ T_1\ \mathsf{else}\ \mathsf{Bag}_d)}{\Gamma \vdash \mathsf{join}(e_1, e_2) : T_3[t \times (t_1 \times t_2)]}$$

$$\frac{\Gamma \vdash e_1 : T[t] \qquad \Gamma \vdash e_2 : T[t] \to T[t] \qquad \Gamma \vdash e : T[t] \to T[t]}{\Gamma \vdash \mu_e\,(e_1, e_2) : T[t]}$$

$$\frac{\forall i\ \Gamma \vdash e_i : t_i}{\Gamma \vdash C(e_1, e_2, ..., e_n) : C[t_1, t_2, ..., t_n]} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash \{\!\{e\}\!\} : \mathsf{Bag}_l[t]}$$

$$\frac{t'_1 + \cdots + t'_n = t' \qquad \mathit{match}(\pi_i, t'_i) \to \Gamma'_i \qquad \Gamma + \Gamma'_i \vdash e_i : t_i \qquad t_1 + \cdots + t_n = t}{\Gamma \vdash \lambda\ \langle \pi_1 \to e_1 \mid \cdots \mid \pi_n \to e_n \rangle : t' \to t}$$

$$\frac{\Gamma \vdash e_1 : t_1 \to t' \qquad \Gamma \vdash e_2 : t_2 \qquad t_2 <: t_1}{\Gamma \vdash e_1\ e_2 : t'} \qquad \frac{\Gamma(a) = t}{\Gamma \vdash a : t} \qquad \frac{\mathit{type}(c) = t}{\Gamma \vdash c : t}$$

Figure 4.3: Typing judgements.

$$\mathsf{flmap}(f, A) = \biguplus_{a \in A} f(a) \qquad \mathsf{reduce}(\otimes, e_\otimes, A) = \bigotimes_{a \in A} f(a)$$

$$\mathsf{reduceByKey}(\otimes, A) = \{\!\{(k, \bigotimes_{(k,v) \in A} v) \mid k \in \mathsf{keys}(A)\}\!\}$$

$$\mathsf{join}(A, B) = \{\!\{(k, (v, w)) \mid (k, v) \in A \wedge (k, w) \in B\}\!\}$$

$$\mathsf{cogroup}(A, B) = \{\!\{(k, (\{\!\{v \mid (k, v) \in A\}\!\}, \{\!\{w \mid (k, w) \in B\}\!\})) \mid k \in \mathsf{keys}(A) \cup \mathsf{keys}(B)\}\!\}$$

where $\mathsf{keys}(A) = \mathsf{distinct}(\{k \mid (k, a) \in A\})$; and $\cup$ is distinct union of bags.

$$\mu_\delta\,(R, \varphi) = \begin{cases} S_N & \text{if there exists N such that } S_{N+1} = S_N \\ \omega & \text{otherwise} \end{cases}$$

$$\text{where } S_n = \bigotimes_{k \leq n}{}_\delta (\delta \circ \varphi)^k(\delta(R))$$

Figure 4.4: Denotational semantics

flatmap has a restriction relative to distributed bags, which is why we also have join; so the only monoid homomorphisms which cannot be written in the form $\lambda\,\langle X \to \mathrm{H}(X)\rangle$ are functions which manipulate distributed bags in a way which cannot be expressed as a join. Thus, it makes sense to check statically whether the term provided by the programmer is of that form and issue a warning if it is not.

4.2.4 μ-monoids denotational semantics

Figure 4.4 gives the denotational semantics of the main algebraic operations. It assumes all terms are well-typed *and* satisfy the additional restrictions mentioned in Sec. 4.2.3. Each closed term has a denotation in the domain corresponding to its type, with the additional possible denotation ω which belongs to all types and represents a computation which does not terminate. Any of those operations returns ω when applied to ω.

These operations, except μ, are monoid homomorphisms [Fegaras, 2017], as discussed in 4.2.2. We can check that they are still monoid homomorphisms if we add ω to all the monoids as an absorbing element.

Properties of μ Recall that δ and φ being compatible means that $\delta \circ \varphi$ is a monoid homomorphism: $M_\delta \to M_\delta$. Thus we have:

$$(\delta \circ \varphi)(\bigotimes_{k \leq n}{}_\delta (\delta \circ \varphi)^k(\delta(R))) = \bigotimes_{k \leq n}{}_\delta (\delta \circ \varphi)^{k+1}(\delta(R)).$$

The only term missing to obtain S_{n+1} on the right is $(\delta \circ \varphi)^0(\delta(R))$, i. e. $\delta(R)$. So we have, for any n: $S_{n+1} = \delta(R) \otimes_\delta (\delta \circ \varphi(S_n))$. In other words, if we use the

definitions to 'clean up' superfluous δs: $S_{n+1} = \Psi(S_n)$ with $\Psi = X \mapsto \delta(R \uplus \varphi(X))$. So S_{n+1} depends only on S_n, making the definition in Fig. 4.4 consistent (if an N is reached such that $S_{N+1} = S_N$ then the sequence becomes stationary) and meaning that $\mu_\delta(R, \varphi)$ is a fixpoint[4] of Ψ.

We now prove the following theorem, which is crucial for optimizing distributed fixpoint computations:

Theorem 1. *$\mu_\delta(\cdot, \varphi)$ is a monoid homomorphism from $\mathsf{Bag}[t] \cup \{\omega\}$ to $M_\delta \cup \{\omega\}$.*

Proof. $\mu_\delta(\{\!\{\}\!\}, \varphi) = \{\!\{\}\!\}$ is immediate.

Let $R_0 = R_1 \uplus R_2$. For all n and for i in $0, 1, 2$, we write $S_n^i = \bigotimes_\delta^{k \le n} (\delta \circ \varphi)^k(\delta(R_i))$. Since δ is a homomorphism from bags to M_δ and $\delta \circ \varphi$ is a homomorphism from M_δ to M_δ, we have $S_n^0 = S_n^1 \otimes_\delta S_n^2$ for all n. Thus, if (S_n^1) and (S_n^2) both become stationary at some point, say N_1 and N_2, then (S_n^0) becomes stationary at $max(N_1, N_2)$ and we do have $\mu_\delta(R_0, \varphi) = \mu_\delta(R_1, \varphi) \otimes_\delta \mu_\delta(R_2, \varphi)$. □

4.2.5 Examples

We present in this section examples of recursive programs expressed in μ-monoids.

Transitive closure (TC)

$$\mu(R, \lambda\, \langle X \to \mathsf{flmap}(\lambda\, \langle (b,(a,c)) \to \{\!\{(a,c)\}\!\}\rangle, \mathsf{join}(\mathsf{flmap}(\lambda\, \langle (a,b) \to \{\!\{(b,a)\}\!\}\rangle, X), R))\rangle)$$

where R is a dataset of tuples (source, destination) representing the edges of a graph.

This expressions computes the entire transitive closure of the input graph R.

The sub-expression $\mathsf{join}(\mathsf{flmap}(\lambda\, \langle (a,b) \to \{\!\{(b,a)\}\!\}\rangle, X), R)$ joins a path from X with a path from R when the target node of the first path corresponds to the start node of the second path. So, at each iteration, the paths in X obtained in the last iteration get appended with edges from R whenever possible. The computation ends when no new paths are found.

Shortest path (SP)

$$\begin{aligned}&\mathsf{reduceByKey}(\min,\\ &\quad \mu(R, \lambda\, \langle X \to \mathsf{flmap}(\lambda\, \langle (b,((a,l_1),(c,l_2))) \to \{\!\{((a,c), l_1 + l_2)\}\!\}\rangle,\\ &\qquad \mathsf{join}(\mathsf{flmap}(\lambda\, \langle ((a,b),l_1) \to \{\!\{(b,(a,l_1))\}\!\}\rangle, X), \mathsf{flmap}(\lambda\, \langle (b,c,l_2) \to \{\!\{(b,(c,l_2))\}\!\}\rangle, R)))))))\end{aligned}$$

where R is a dataset of tuples (source, destination, weight) representing the weighted edges of a graph.

The expression computes the shortest path between each pair of nodes in the input graph R. New paths are computed by performing a transitive closure while summing the lengths of the joined paths. Finally, the $\mathsf{reduceByKey}$ operation keeps the shortest paths between each pair of nodes.

[4] The least fixpoint, if we define an appropriate ordering relation on M_δ, e. g. set inclusion in the standard case where $\delta = \mathsf{distinct}$.

Flights

$$\mu(R, \lambda\ \langle X \rightarrow$$
$$\mathsf{flmap}(\lambda\ \langle(\mathsf{corr}, (\mathsf{Flight}(\mathsf{dtime1}, \mathsf{atime1}, \mathsf{dep1}, \mathsf{dest1}, \mathsf{dur1}), \mathsf{Flight}(\mathsf{dtime2}, \mathsf{atime2}, \mathsf{dep2}, \mathsf{dest2}, \mathsf{dur2}))) \rightarrow$$
$$\mathsf{if}\ \mathsf{atime1} < \mathsf{dtime2}\ \mathsf{then}\ \{\!\{\mathsf{Flight}(\mathsf{dtime1}, \mathsf{atime2}, \mathsf{dep1}, \mathsf{dest2}, \mathsf{dur1} + \mathsf{dur2})\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle,$$
$$\mathsf{join}(\mathsf{flmap}(\lambda\ \langle\mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{dep}, \mathsf{corr}, \mathsf{dur}) \rightarrow (\mathsf{dest}, \mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{dep}, \mathsf{corr}, \mathsf{dur}))\rangle, X)),$$
$$\mathsf{flmap}(\lambda\ \langle\mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{corr}, \mathsf{dest}, \mathsf{dur}) \rightarrow (\mathsf{dep}, \mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{corr}, \mathsf{dest}, \mathsf{dur}))\rangle, R))))))$$

where R is a dataset of direct flights. $\mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{dep}, \mathsf{dest}, \mathsf{dur})$ is a flight object with a departure time dtime, arrival time atime, departure location dep, destination dest and duration dur. At each iteration, the fixpoint expression computes new flights by joining the flights obtained at the previous iteration with the flights dataset, in such a way that two flights produce a new flight if the first flight arrives before the second flight departs, and the first flight destination airport is the second's flight departure airport. The computation stops when no more new non-direct flights can be deduced.

Path planning

$$\mathsf{flmap}(\lambda\ \langle((s, d), l) \rightarrow \mathsf{if}\ s = \text{"Paris"}\ \text{and}\ d = \text{"Geneva"}\ \mathsf{then}\ \{\!\{\mathsf{Path}(s, d, l)\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle,$$
$$\mathsf{reduceByKey}(\mathsf{bestRated}, \mathsf{flmap}(\lambda\ \langle(s, d, l) \rightarrow ((s, d), l))\rangle, F)))$$

$$F = \mu(R, \lambda\ \langle X \rightarrow \mathsf{flmap}(\lambda\ \langle(k, ((s, l_1), (d, l_2))) \rightarrow \{\!\{(s, d, l_1 +\!\!+ l_2)\}\!\}\rangle, \mathsf{join}($$
$$\mathsf{flmap}(\lambda\ \langle(s, k, l) \rightarrow \{\!\{(k, (s, l))\}\!\}\rangle, X),$$
$$\mathsf{flmap}(\lambda\ \langle(\mathsf{City}(k, l_1), \mathsf{City}(d, l_2)) \rightarrow (k, (d, l_2)))\rangle, R)))))$$

where R is a set of routes between two cities. Each city $\mathsf{City}(n, l)$ has a name n and a set of landmarks l and each landmark $\mathsf{Landmark}(n, r)$ has a rating r. $\mathsf{bestRated}(l_1, l_2)$ is a function that returns the best set of landmarks based on its ratings.

The fixpoint F computes the set of landmarks that can be visited for each possible path between each two cities. The final term then computes the best path between Paris and Geneva.

Movie Recommendations

$$\mu(S, \lambda\ \langle X \rightarrow \mathsf{flmap}(\lambda\ \langle x \rightarrow \mathsf{flmap}(\lambda\ \langle\mathrm{User}(u, bm) \rightarrow \mathsf{if}\ x \in bm\ \mathsf{then}\ bm\ \mathsf{else}\ \{\!\{\}\!\}\rangle, U)\rangle, X)\rangle)$$

where U is a set of users, each user $\mathrm{User}(u, bm)$ has a set of best movies bm.

The query computes a set of recommended movies by starting from a set of movies S and by adding the best movies of a user if one of his best movies is in the set of recommended movies until no new movie is added.

4.2.6 Evaluation of expressions

Local execution

Pattern matching and function application The result of matching a value against a pattern is either a set of pattern variable assignments or $\perp$. It is defined

as follows:

$$\begin{aligned} \mathrm{m}(v, a) &= \{a \mapsto v\} \\ \mathrm{m}(C(v_1, ..., v_n), C(\pi_1, ..., \pi_n)) &= \mathrm{m}(v_1, \pi_1) \cup \cdots \cup \mathrm{m}(v_n, \pi_n) \\ \mathrm{m}(C(\cdots), C'(\cdots)) &= \bot \text{ if } C \neq C' \end{aligned}$$

where we extend $\cup$ so that $\bot \cup S = \bot$.

A lambda expression $f = \lambda \langle \pi_1 \to e_1 \mid \cdots \mid \pi_n \to e_n \rangle$ contains a number of patterns together with return expressions. When this lambda expression is applied on an argument v ($f\ v$), the argument is matched against the patterns in order, until the result of the match is not $\bot$. Let i be the smallest index such that $\mathrm{m}(v, \pi_i) = S \neq \bot$, the result of the application is obtained by substituting the free pattern variables in e_i according to the assignments in S.

Monoid homomorphisms The definition of algebraic operations as monoid homomorphism suggests that they can be evaluated in the following way: if φ is a homomorphism from $\mathsf{Bag}[t]$ to $(t', e, \otimes)$, $\varphi(\{\!\{v_1 ... v_n\}\!\}) \rightsquigarrow \varphi(\{\!\{v_1\}\!\}) \otimes \varphi(\{\!\{v_2\}\!\}) \otimes ... \otimes \varphi(\{\!\{v_n\}\!\})$. As monoid operators are associative, parts of an expression in the form $e_1 \otimes e_2 \otimes ... \otimes e_n$ can be evaluated in any order and in parallel.

Fixpoint operator The fixpoint operator can be evaluated as a loop, as described in Def. 10. We can summarise it with the following reduction rules, where $R\mu$ represents a running μ computation and has the bag which accumulates the results as an additional parameter:

$$\mathcal{R}_{init}\ \mu_\delta\,(R, \varphi) \rightsquigarrow R\mu_\delta\,(\delta(R), \varphi; \delta(R)) \qquad \mathcal{R}_{stop}\ \frac{S \otimes_\delta \varphi(R) = S}{R\mu_\delta\,(R, \varphi; S) \rightsquigarrow S}$$

$$\mathcal{R}_{loop}\ \frac{S \otimes_\delta \varphi(R) \neq S}{R\mu_\delta\,(R, \varphi; S) \rightsquigarrow R\mu_\delta\,(\delta(\varphi(R)), \varphi; S \otimes_\delta \varphi(R))}$$

Distributed execution

We consider in a distributed setting that distributed bags are partitioned. Distributed data is noted in the following way: $R = R_1|R_2|...|R_p$, meaning that R is split into p partitions stored on p machines. We can write a new slightly different version of the rule described above for evaluating partitioned data:

- $\varphi(R_1|R_2|...|R_p) \rightsquigarrow \varphi(R_1)|\varphi(R_2)|...|\varphi(R_p)$ if φ is an homomorphism from bags to bags (partitioning does not have to change)
- $\varphi(R_1|R_2|...|R_p) \rightsquigarrow \varphi(R_1) \otimes^{nl} \varphi(R_2) \otimes^{nl} ... \otimes^{nl} \varphi(R_p)$ if φ is an homomorphism from bags to $(M, e, \otimes)$, where $\otimes^{nl}$ is the non-local version of $\otimes$. Applying this non-local operation means that data transfers are required.

This means that in our algebra, all operators apart from flatmap, or join when one of the parameters is a local bag, need to send data across the network (for executing the non-local version of their monoid operator). The execution of these non-local operators depends on the distributed platform. Spark for example performs *shuffling* to redistribute the data across partitions for the computation of certain of its operations like cogroup and groupByKey.

4.3 Optimizations

In this section, we propose new optimization rules for terms with fixpoints, and describe when and how they apply. The purposes of the rules are (i) to identify which basic operations within an algebraic term can be rearranged and under which conditions, and (ii) to describe how new terms are produced or evaluated after transformation.

We first give the intuition behind each optimization rule before zooming on each of them to formally describe when they apply. The four new optimization rules are:

- PF is a rewrite rule of the form:

$$F(\mu(R, \varphi)) \longrightarrow \mu(F(R), \varphi)$$

  it aims at pushing a filter F inside a fixpoint, whenever this is possible. A filter is a function which keeps only some elements of a dataset based on their values; we define it formally in Sec. 4.3.1.

- PJ is a rewrite rule of the form:

$$\mathsf{join}(A, \mu(R, \varphi)) \longrightarrow \mathsf{join}(A, \mu(F_A(R), \varphi))$$

  it aims at inserting a filter F_A inside a fixpoint before a join is performed. It is inspired by the semi-join found in relational databases, and tailored for μ-monoids.

- PA is a rewrite rule of the form:

$$\delta(\mu(R, \varphi)) \longrightarrow \mu_\delta\,(R, \varphi)$$

  It aims at pushing an aggregation function δ inside a fixpoint, transforming a simple fixpoint into a fixpoint with aggregation. This rule requires δ to be compatible with φ; it is inspired from the premappability condition in Datalog [Zaniolo et al., 2017].

- an optimization rule $\mathrm{P}_{\mathrm{dist}}$ that determines how a fixpoint term is evaluated in a distributed manner by choosing among two possible execution plans.

4.3.1 Pushing filter inside a fixpoint (PF)

Filter depending on a single pattern variable

Definition 15 (filter)**.** *We call* filter *a function of the form:*

$$\lambda\ \langle D \to \mathit{flmap}(\lambda\ \langle \pi \to \mathit{if}\ c(a)\ \mathit{then}\ \{\!\{\pi\}\!\}\ \mathit{else}\ \{\!\{\}\!\}\rangle, D)\rangle$$

where π is a pattern containing the variable a and $c(a)$ is a Boolean condition depending on the value of a.

Such a function returns the dataset D filtered by retaining only the elements whose value for a (as determined by pattern-matching that element with π) satisfies $c(a)$. The elements are unmodified, so the result is a subcollection of D.

In the following, we consider a filter F with π and a defined as above, and we denote by π_a the function that matches an element against π and returns the value of a ($\pi_a = \lambda\ \langle \pi \to a\rangle$). For instance, $\pi_a((1,(5,6))) = 5$ for $\pi = (x,(a,y))$.

Let us consider a dataset D. In terms of denotational semantics, with the notations above, we have $F(D) = \{d \in D \mid c(\pi_a(d))\}$.

The PF rule This rule consists in transforming an expression of the form $F(\mu(R,\varphi))$ to an expression of the form $\mu(F(R),\varphi)$, where F is a filter.

In the second form, the filter is pushed before the fixpoint operation. In other words, the constant part R is filtered first before applying the fixpoint on it. We now present sufficient conditions for the two terms to be equivalent.

PF condition Let (C) be the following condition:

$$\forall r \in R \quad \forall s \in \varphi(\{\!\{r\}\!\}) \quad \pi_a(r) = \pi_a(s)$$

Intuitively, this condition means that the operation φ does not change the part of its input data that corresponds to a in the pattern π, which is the part used in the filter; so for each record in the fixpoint that does not pass the filter, the record in R that has originated it does not pass the filter and the other way round. That is why we can just filter R in the first place.

Let $A = \mathsf{flmap}(\lambda\ \langle \pi \to \mathsf{if}\ c(a)\ \mathsf{then}\ \{\!\{\pi\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle, \mu(R,\varphi))$. We prove that if (C) is satisfied, then $A = \mu(F(R),\varphi)$. To prove this, we use the following property of fixpoints where $\delta = \mathsf{distinct}$:

Lemma 4. $\forall a \in \mu(R,\varphi) \quad a \in \varphi^{(n)}(\{\!\{r\}\!\})$ *for some $r \in R$ and $n \in N$*

Proof. We have:

$$\mu(R,\varphi) = \bigcup_{n\in\mathbb{N}} \varphi^{(n)}(R) = \bigcup_{n\in\mathbb{N}} \varphi(\biguplus_{r\in R} \{\!\{r\}\!\}) = \bigcup_{n\in\mathbb{N}} (\biguplus_{r\in R} \varphi(\{\!\{r\}\!\})) = \bigcup_{n\in\mathbb{N}} (\bigcup_{r\in R} \varphi(\{\!\{r\}\!\}))$$

□

Using the above lemma and condition (C), we have:

$$(*) \qquad \forall s \in \mu(R,\varphi) \quad \exists r \in R \quad \pi_a(r) = \pi_a(s)$$

We now prove $A = \mu(F(R),\varphi)$ by proving the two inclusions:

1. $\mu(F(R), \varphi) \subset A$:

   $F(R) \subset R \Rightarrow \mu(F(R), \varphi) \subset \mu(R, \varphi)$ (because $\mu(R, \varphi) = \mu(F(R) \uplus R', \varphi) = \mu(F(R), \varphi) \cup \mu(R', \varphi)$ and $\mu(R, \varphi)$ does not contain duplicates)

   Let $s \in \mu(F(R), \varphi)$

   $\exists r \in F(R) \quad \pi_a(s) = \pi_a(r) \quad (*)$

   So $c(\pi_a(s)) = c(\pi_a(r)) = \text{true}$ (because $r \in F(R)$)

   So $s \in \mu(R, \varphi)$ and $c(\pi_a(s))$ is true, then $s \in A$

2. $A \subset \mu(F(R), \varphi)$:

   Let $s \in A$. We have: $s \in \mu(R, \varphi)$ and $c(\pi_a(s)) = \text{true}$

   So $\exists r \in R \quad \exists n \in \mathbb{N} \quad \pi_a(s) = \pi_a(r)$ (because $(*)$ and $s \in \varphi^{(n)}(\{\!\{r\}\!\})$)

   So $c(\pi_a(r)) = c(\pi_a(s)) = \text{true}$

   So $r \in F(R)$, which means $\mathsf{distinct}(\varphi^{(n)}(\{\!\{r\}\!\})) \subset \mu(F(R), \varphi)$, so $s \in \mu(F(R), \varphi)$.

Verifying the condition (C) using type inference We will start by explaining the intuition behind this before going into the details.

For the condition (C) to hold, we need to make sure that the part of the data extracted by π_a is not modified by φ. For this, our solution is inspired by the idea that the type of a parametric polymorphic function tells us information about its behaviour [Wadler, 1989]: for example, if f is a polymorphic function whose argument contains exactly one value of the undetermined type α and whose result must also contain a value of type α, then the α value in the result is necessarily the one in the argument ($f : \alpha \to \alpha \Rightarrow \forall x\ f(x) = x$).

This reasoning can also be used for a more complex input type $C(\alpha)$ that contains a polymorphic type α. For instance: $C(\alpha) = A(B(\alpha), D)$ is such a type given that A,B and D are type constuctors. So our goal is, given that φ takes as input a bag of elements of type C, to find an appropriate polymorphic type $C(\alpha)$ that will be used for type checking φ. In practice, we translate the φ operation to a Scala function that takes a polymorphic input type and use the Scala type inference system [Odersky et al., 2004] to get the output type[5]. $C(\alpha)$ should be built in such a way that the position of α in $C(\alpha)$ is the same as the position of a in π. Such a type is possible to build because the type C matches the pattern π, otherwise the filtered term would not be type correct. Finally, if the output type also contains the type α and has the same position as a in π then we can show that the condition (C) holds. Note that we do not need a full-fledged parametricity theorem for this: we only use the fact that the Scala type system has singleton types for all values.

[5] We consider it a more practical solution than implementing our own type inference system supporting polymorphism.

Building $C(\alpha)$ Types are made from type constructors and basic types, and patterns are made from type constructors and pattern variables. So we can represent their structures using trees. In the following we sometimes refer to types by the trees representing them.

Definition 16 (path)**.** *We define the* path *to the node labelled n in the tree T denoted $path(n, T)$ by the ordered sequence $Seq(a_i)$ where a_i is the next child arity of the ith visited node to reach n from the root of the tree. A node in a tree can be identified by its path.*

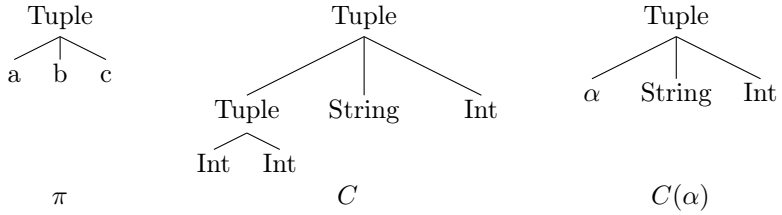
Let us consider the function $\texttt{replace}_\alpha(p, T)$ that, given a path p and a type T returns a polymorphic type $T(\alpha)$ that is obtained by replacing in T the node at path p and its children by a node labelled α. Let us now consider $C(\alpha) = replace_\alpha(path(a, \pi), C)$, where $\mathsf{Bag}[C]$ is the input type of φ. Note that this path makes sense in C because C matches π (see Appendix 4.2.3).

With $C(\alpha)$ built this way, we have the following:

$$e : C \text{ and } \pi_a(e) : \alpha \Rightarrow e : C(\alpha) \tag{4.1}$$

$$e : C(\alpha) \Rightarrow \pi_a(e) : \alpha \tag{4.2}$$

For example:



We show that if $\varphi : \mathsf{Bag}[C(\alpha)] \to \mathsf{Bag}[C(\alpha)]$ then the condition (C) is verified:

Let $r \in R$ and let us take $\alpha = \{\!\{\pi_a(r)\}\!\}$ which is the singleton type containing the value $\pi_a(r)$. Since $\pi_a(r) : \alpha$ and $r : C$, we have $r : C(\alpha)$ according to (1).

We also have $\varphi(\{\!\{r\}\!\}) : \mathsf{Bag}[C(\alpha)]$ because $\{\!\{r\}\!\} : \mathsf{Bag}[C(\alpha)]$ and $\varphi : \mathsf{Bag}[C(\alpha)] \to \mathsf{Bag}[C(\alpha)]$ which means that $\forall s \in \varphi(\{\!\{r\}\!\}) \quad s : C(\alpha)$. So $\pi_a(s) : \alpha$ (according to (2)) so $\pi_a(s) = \pi_a(r)$ hence (C).

Filters depending on multiple variables

We showed that (C): $\forall r \in R \quad \forall s \in \varphi(\{\!\{r\}\!\}) \quad \pi_a(r) = \pi_a(s)$ is sufficient for pushing the filter in a fixpoint when the filter condition depends on a. We can easily show that when the filter depends on a set of pattern variables V, the sufficient condition becomes: $\forall r \in R \quad \forall s \in \varphi(\{\!\{r\}\!\}) \quad \forall v \in V \quad \pi_v(r) = \pi_v(s)$. So, if one of the variables in V does not satisfy the condition the filter would not be pushed. However, we can do better by trying to split the condition c to two conditions c_1 and c_2, such that $c = c_1 \wedge c_2$ and c_1 depends only on the subset of variables that satisfies the condition (this splitting technique is used in [Fegaras and Noor, 2018] to push filters in a cogroup or a groupby). If such a split is found, the filter $\mathsf{flmap}(\lambda\ \langle \pi \to \mathsf{if}\ c\ \mathsf{then}\ \{\!\{\pi\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle, R)$ can be rewritten as $\mathsf{flmap}(\lambda\ \langle \pi \to \mathsf{if}\ c_2\ \mathsf{then}\ \{\!\{\pi\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle, \mathsf{flmap}(\lambda\ \langle \pi \to \mathsf{if}\ c_1\ \mathsf{then}\ \{\!\{\pi\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle, R))$. The inner filter can then be pushed.

4.3.2 Filtering inside a fixpoint before a join (PJ)

Let us consider the expression: $\mathsf{join}(A, B)$, where A is a constant and $B = \mu(R, \varphi)$. After the execution of the fixpoint, the result is going to be joined with A, so only elements of this result sharing the same keys with A are going to be kept. So in order to optimize this term, we want to push a filter that keeps only the elements having a key in A. This way, elements not sharing keys with A are going to be removed before applying the fixpoint operation on them.

1. we show that $join(A, B) = join(A, F_A(B))$, where $F_A(B) = \{(k, v) \mid (k, v) \in B, \exists w\ (k, w) \in A\}$:

   We have $join(A, B) = \{(k, (x, y)) \mid (k, x) \in A, (k, y) \in B\}$.

   So $(k, (x, y)) \in join(A, B) \Leftrightarrow (k, x) \in A \wedge (k, y) \in B \Leftrightarrow (k, x) \in A \wedge ((k, y) \in B \wedge \exists w\ (k, w) \in A) \Leftrightarrow (k, x) \in A \wedge (k, y) \in F_A(B) \Leftrightarrow (k, (x, y)) \in join(A, F_A(B)))$.

2. we show that $F_A(B)$ is a filter on B. This filter can be pushed when the criteria on pushing filters is fulfilled:

   We can show that $F_A(B) = \mathsf{flmap}(\lambda\ \langle (k, v) \rightarrow \mathsf{if}\ c(k)\ \mathsf{then}\ \{\!\{(k, v)\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle, B)$ where $c(k)$ is the boolean expression that corresponds to the predicate $\exists w (k, w) \in A$. This expression can be: $c(k) = \mathsf{reduce}(\vee, e_\vee, \mathsf{flmap}(\lambda\ \langle (k', a) \rightarrow k == k'\rangle, A))$. Which means that in case φ fulfills the criteria for pushing filters we will have $F_A(B) = F_A(\mu(R, \varphi)) = \mu(F_A(R), \varphi)$.

3. we show as well that $F_A(B) = C$ where:

   $C = \mathsf{flmap}(\lambda\ \langle (k, (s_x, s_y)) \rightarrow \mathsf{if}\ s_y \neq \{\!\{\}\!\}\ \mathsf{then}\ \mathsf{flmap}(\lambda\ \langle x \rightarrow \{\!\{(k, x)\}\!\}\rangle, s_x)\ \mathsf{else}\ \{\!\{\}\!\}\rangle,$
   $\mathsf{cogroup}(B, A))$

   We have: $C = \biguplus_{(k,(s_x,s_y)) \in \mathsf{cogroup}(B,A)} \biguplus_{x \in s_x} (\mathsf{if}\ s_y \neq \{\!\{\}\!\}\ \mathsf{then}\ \{\!\{(k, (x, y))\}\!\}\ \mathsf{else}\ \{\!\{\}\!\})$

   (1) Let $e \in C$. So $\exists (k, (s_x, s_y))) \in \mathsf{cogroup}(B, A)$ such that $\exists x \in s_x \quad e = (k, x)$ and $s_y \neq \{\!\{\}\!\}$.

   We have $(k, (s_x, s_y)) \in \mathsf{cogroup}(B, A)$, so $s_x = \{v \mid (k, v) \in B\}$, which means that $(k, x) \in B$ because $x \in s_x$. And $s_y \neq \{\!\{\}\!\}$ means that $\exists w\ (k, w) \in A$, so $e = (k, x) \in F_A(B)$.

   (2) Let $(k, x) \in F_A(B)$. We have $(k, x) \in B$ and $\exists w\ (k, w) \in A$. So $k \in \mathsf{keys}(A) \cup \mathsf{keys}(B)$.

   Let $s_x = \{v \mid (k, v) \in B\}$ and $s_y = \{v \mid (k, y) \in A\}$, so $(k, (s_x, s_y)) \in \mathsf{cogroup}(B, A)$.

   Since $x \in s_x$ and $s_y \neq \{\!\{\}\!\}$ (because $\exists w\ (k, w) \in A$), then $(k, x) \in C$.

4.3.3 Pushing aggregation into a fixpoint (PA)

The PA rule consists in rewriting a term of the form $\delta(\mu(R, \varphi))$ to a term of the form $\mu_\delta\,(R, \varphi)$. It requires that δ is an aggregation function and compatible with φ.

It is correct thanks to the following lemma:

Lemma 5. *Let φ be a monoid homomorphism: $\mathsf{Bag}[t] \to \mathsf{Bag}[t]$, δ an aggregation function: $\mathsf{Bag}[t] \to \mathsf{Bag}[t]$ compatible with φ, and $R : \mathsf{Bag}[t]$ a dataset. Assume $\mu(R, \varphi) \neq \omega$ (i. e. the computation terminates). Then $\delta(\mu(R, \varphi)) = \mu_\delta(R, \varphi)$.*

Proof. Let (S_n) and (S'_n) be the S sequences corresponding respectively to the two fixpoints; thus we have $S_{n+1} = R \uplus \varphi(S_n)$ and $S'_{n+1} = R \otimes_\delta \varphi(S'_n) = \delta(R \uplus \varphi(S'_n))$. We prove by induction on n that $\delta(S_n) = S'_n$ for any n: for $n = 0$ we have $\delta(S_0) = \delta(R) = S'_0$. Assume $S'_n = \delta(S_n)$, we have:

$$\begin{aligned}
\delta(S_{n+1}) &= \delta(R \uplus \varphi(S_n)) = \delta(\delta(R) \uplus \delta(\varphi(S_n))) && (\delta \text{ is an aggregation function}) \\
&= \delta(\delta(R) \uplus \delta(\varphi(\delta(S_n)))) && (\delta \circ \varphi = \delta \circ \varphi \circ \delta) \\
&= \delta(\delta(R) \uplus \delta(\varphi(S'_n))) && (\text{induction hypothesis}) \\
&= \delta(R \uplus \varphi(S'_n)) = S'_{n+1} && (\delta \text{ is an aggregation function})
\end{aligned}$$

Then the result propagates to the fixpoint since we assumed that $\mu(R, \varphi) \neq \omega$. □

Applying this optimization on the expression of the SP example (Sec.4.2.5) means that only the shortest paths are kept at each iteration of the fixpoint so we avoid computing all possible paths before keeping only the shortest ones at the end.

4.3.4 Distribution of the fixpoint operations ($\mathbf{P_{dist}}$)

As explained in 4.2.6, the fixpoint operation is computed locally using a loop (defined by $\mathcal{R}_{init}, \mathcal{R}_{loop}$ and $\mathcal{R}_{stop}$). To evaluate the fixpoint in a distributed setting, we could simply write a loop that distributes the computation of the operation that is performed at each iteration ($S \otimes_\delta \varphi(R)$) among the workers. We call this execution plan $\mathcal{P}_{\texttt{gld}}$. $\mathcal{P}_{\texttt{gld}}$ performs δ at each iteration on the whole intermediary distributed bag S to compute $S \otimes_\delta \varphi(R)$, which in most cases (i. e. unless δ is the identity function) requires synchronisation and data transfer between workers at each iteration. In the TC example (Sec. 4.2.5), this plan amounts to appending, at each iteration, all currently found paths from all partitions with the graph edges R.

Alternatively, if we use the fact that $\mu_\delta(R, \varphi)$ is a monoid homomorphism, then we can replace $\mathcal{R}_{init}$ with the following distributed version (recall that $R_1|R_2|...$ denotes a distributed bag split across different partitions R_i. $\otimes_\delta^{nl}$ denotes the non-local version of $\otimes_\delta$):

$$\mu_\delta(R_1|R_2|..., \varphi) \rightsquigarrow R\mu_\delta(\delta(R_1), \varphi; \delta(R_1)) \otimes_\delta^{nl} R\mu_\delta(\delta(R_2), \varphi; \delta(R_2)) \otimes_\delta^{nl} ...$$

Then each $R\mu_\delta(\delta(R_i), \varphi; \delta(R_i))$ is going to be evaluated by $\mathcal{R}_{loop}$ and $\mathcal{R}_{stop}$ as they are fixpoints on local bags. This execution plan, that we name $\mathcal{P}_{\texttt{plw}}$, will avoid doing non-local set unions or aggregations between all partitions at each iteration of the fixpoint. Instead, the fixpoint is executed locally on each partition on a part of the input, after which the aggregate $\otimes_\delta^{nl}$ is computed once to gather results. In our example, this amounts to computing, on each partition i, all paths

in the graph starting from nodes in R_i; the result is then the union of all obtained paths.

This reduction in data transfers can lead to a significant improvement of performance, since the size of data transfers over the network is a determining factor of the performance of distributed applications.

The optimization rule $\mathrm{P}_{\mathrm{dist}}$ uses the plan $\mathcal{P}_{\mathtt{plw}}$ instead of $\mathcal{P}_{\mathtt{gld}}$ for evaluating fixpoints.

Avoiding $\cup^{nl}$ in $\mathcal{P}_{\mathtt{plw}}$ In the common case where δ is $\mathsf{distinct}$, $\mathcal{P}_{\mathtt{plw}}$ can be optimized further by repartitioning the data in the cluster in such a way that every result of the fixpoint appears in one partition only. When that is the case, it is sufficient to perform a bag union rather than a set union that removes duplicates from across the cluster. If we know that there is a part in the input that does not get modified by φ, we can repartition the data on this part of the input (no two different partitions have the same value for this part), so the result of the fixpoint is also going to be repartitioned in the same way. We formalize this optimization in the following way:

Let π a pattern that matches the input of φ and a a pattern variable in π. We consider the following propositions:

$$(C_a) : \forall r \in R \quad \forall s \in \varphi(\{\!\{r\}\!\}) \quad \pi_a(r) = \pi_a(s)$$
$$(P_a) : \forall i \neq j \; \forall x \in R_i \; \forall y \in R_j \quad \pi_a(x) \neq \pi_a(y)$$

Lemma 6. *If there exists a pattern variable a that verifies (C_a), then:*

$$P_a \Rightarrow \forall i \neq j \quad \mu(R_i, \varphi) \cap \mu(R_j, \varphi) = \emptyset$$

Proof. Let us suppose there exist a pattern variable a for which (C_a) is verified, and let us suppose (P_a). Let R_i and R_j partitions of R such that $i \neq j$. (C_a) implies $\forall s \in \mu(R, \varphi) \quad \exists r \in R \quad \pi_a(r) = \pi_a(s)$ because of Lemma 4. Which means that for any $x \in \mu(R_i, \varphi)$ and $y \in \mu(R_j, \varphi)$, $\exists r_i \in R_i \; \exists r_j \in R_j \quad \pi_a(r_i) = \pi_a(x)$ and $\pi_a(r_j) = \pi_a(y)$. We have $\pi_a(r_i) \neq \pi_a(r_j)$ because (P_a), so $x \neq y$. Hence $\forall i \neq j \;\; \mu(R_i, \varphi) \cap \mu(R_j, \varphi) = \emptyset$. □

This means that:

$$\mu(R_1 \cup \varphi(R_1), \varphi) \cup^{nl} \mu(R_2 \cup \varphi(R_2), \varphi) \cup^{nl} ... = \mu(R_1 \cup \varphi(R_1), \varphi) \mid \mu(R_2 \cup \varphi(R_2), \varphi) \mid ...$$

The pattern variable a that verifies (C_a) can be found by using the technique explained in Section 4.3.1. We explore every node n in C ($\mathsf{Bag[C]}$ is the input type of φ) starting from the root of C and we build $C(\alpha) = \mathsf{replace}_{\alpha}(\mathsf{path}(n, C), C)$ until we find a node that verifies $\varphi : C(\alpha) \to C(\alpha)$.

If such a is found, we repartition the data according to (P_a) by using the API provided by the big data platform on which the code is executed, given that a can be extracted from the input data using pattern matching.

4.3.5 Rule application criteria

Rule PF is a logical optimization rule in the sense that the term it produces is always more efficient than the initial term. Indeed, a filter reduces the size of intermediate data. The application of PF thus reduces data transfers. Operators are also executed faster on smaller data. The application of PF can thus only improve performance.

The rest of the rules however require specific criteria to ensure that their application actually enhances performance.

Criteria for PJ

The rule PJ introduces an additional cogroup to compute the filter being pushed in the fixpoint (as detailed in Sec. 4.3.2). To estimate the cost of evaluating a term, two important aspects are considered: the size of non-local data transfers it generates, and the local complexity of the term (i.e. the time needed for executing its local operations). PJ can improve local complexity. The reason is that the additional cogroup is evaluated only once, whereas the pushed filter makes R (the first argument of the fixpoint $\mu\,(R, \varphi)$) smaller. Therefore, in general, each iteration of the fixpoint is executed faster as it deals with increasingly less data (each value removed from the initial bag would have generated more additional values with each iteration). The final join with the result of the fixpoint also executes faster because its size is reduced prior to the join. We can then consider that, in general, the additional cogroup cost is compensated by the speedup of each iteration in the fixpoint as well as the final join. To analyse the impact of the rule on non-local data transfers, we estimate and compare the size of transfers incurred by the terms: $join(A, \mu(R, \varphi))$ and $join(A, \mu(F_A(R), \varphi))$ (obtained after applying the rule). As mentioned in Section 4.2.6, all our algebraic operators apart from flatmap trigger non-local transfers. We then consider the following, where $\mathsf{size}_t(e)$ is the size of transfers incurred by the term e:

- We assume that $\mathsf{groupby}(A)$ incurs a transfer size that is linear to the size of A (all A needs to be sent to be seen by other partitions). So, $\mathsf{size}_t(\mathsf{groupby}(A)) \approx o(\mathsf{size}(A))$
- Similarly, $\mathsf{cogroup}(A,\, B)$ and $\mathsf{join}(A,\, B)$ transfer A and B (the cogroup and join operators are made between all elements of A and B):
  $\mathsf{size}_t(\mathsf{cogroup}(A, B)) \approx o(\mathsf{size}(A) + \mathsf{size}(B))$
  $\mathsf{size}_t(\mathsf{join}(A, B)) \approx o(\mathsf{size}(A) + \mathsf{size}(B))$
- $\mu(R, \varphi)$ would have to send all its result in order to compute the set union operation. So we just refer to $\mathsf{size}(\mu(R, \varphi))$ to indicate the size of the fixpoint result.

Let $S_1 = \mathsf{size}_t(\mathsf{join}(A, \mu(R, \varphi)))$
and $S_2 = \mathsf{size}_t(\mathsf{join}(A, \mu(F_A(R), \varphi)))$

$S_1 \approx o(\mathsf{size}(A) + 2 \times \mathsf{size}(\mu(R, \varphi)))$, here the result of the fixpoint is sent twice: the first time to compute the fixpoint and the second time to compute the join between A and the fixpoint result.

$S_2 \approx o(2 \times \mathsf{size}(A) + 2 \times \mathsf{size}(\mu(F_A(R), \varphi)) + \mathsf{size}(R))$, here $F_A(R)$ requires making a cogroup between A and R which incurs an additional transfer of their sizes. On the other hand, only a filtered fixpoint result is sent.

In order to determine if PJ improves data transfers we compare S_1 and S_2, which amounts to comparing the following quantities: $2 \times \mathsf{size}(\mu(R, \varphi))$ and $2 \times \mathsf{size}(\mu(F_A(R), \varphi)) + \mathsf{size}(A) + \mathsf{size}(R)$. In other words, this estimates whether the data removed from the fixpoint result (by pushing the filter into it) makes up for the sizes of A and R that are transferred to compute the additional cogroup.

Criteria for PA

The PA rule applies the aggregation function δ on the fixpoint's intermediate results instead of once at the end. Usually, δ reduces the size of these results. This means that the fixpoint operation deals with less data at each iteration (which also generally reduces the number of iterations). For example, if we are computing the shortest paths, applying the rule would mean that we are only going to deal with the shortest paths at each step instead of the entirety of possible paths. This can also lead to the termination of the program in case the graph has cycles (note that the programs are semanticaly equivalent but the evaluation of the first does not terminate). Additionally, when $\mathrm{P}_{\mathrm{dist}}$ is applied, PA can only reduce the size of the data transferred across the network because δ is executed locally and reduces the sizes of the local fixpoints.

To be applicable, this optimization requires that δ is an aggregation function and is compatible with φ. The latter constraint means that the application of δ first before the φ operation does not impact the result compared to when it is applied once at the end. For instance, if we are computing the shortest paths between a and b, we look for all paths between a and c, append them to paths from c to b, then keep the shortest ones. Alternatively, we could start by keeping only the shortest paths between a and c then append them to paths between c and b without altering results.

At present, we do not have a method for statically checking this constraint. However, we can list common known aggregation functions: $\mathsf{reduceByKey}(f, \cdot)$, filters (see Def. 15), mainly. We can also envision for future work to define sufficient conditions for a φ to be compatible with specific known aggregation functions such as e. g. $\mathsf{reduceByKey}(min, \cdot)$, so that the optimization can take place automatically in common use cases.

Criteria for $\mathrm{P}_{\mathrm{dist}}$ in the Spark setting

The application of the rule $\mathrm{P}_{\mathrm{dist}}$ can exploit platform-specific criteria. For instance, for Spark [Zaharia et al., 2016], the choice between plans $\mathcal{P}_{\mathtt{gld}}$ and $\mathcal{P}_{\mathtt{plw}}$ is parameterized based on two key aspects. First, for a term $\mu(R, \varphi)$, the collections referenced in φ have to be available locally in each worker so that it can compute the fixpoint locally. For instance, if $\varphi = join(X, S)$ then S and X (at each iteration) are both referenced by φ. This is a limitation of plan $\mathcal{P}_{\mathtt{plw}}$: when those datasets become too large to be handled by one worker, $\mathcal{P}_{\mathtt{gld}}$ is favored. Second, in Spark, a factor that determines the efficiency of $\mathcal{P}_{\mathtt{plw}}$ is the number of

partitions used by the program. Increasing the number of partitions increases the parallelization and reduces the load on each worker because the local fixpoints start from smaller constant parts. For a term $\mu(R, \varphi)$, it is thus possible to regulate the load on the workers by splitting R into smaller R_i, resulting in smaller tasks on more partitions. The ideal number of partitions is the smallest one that makes all workers busy for the same time period, and for which the size of the task remains suitable for the capacity of each worker. Increasing the number of partitions further would only increase the overhead of scheduling. Thus, before choosing plan $\mathcal{P}_{\texttt{plw}}$, the rule $\mathrm{P}_{\mathrm{dist}}$ estimates an appropriate number of partitions, based on an estimated size of the constant part, the size of intermediate data produced by the fixpoint and the workers memory capacity.

4.4 Experimental results

Methodology. We experiment the μ-monoids approach in the context of the Spark platform [Zaharia et al., 2016].

We evaluate Spark programs generated from optimized μ-monoids expressions, and compare their performance with the state-of-the-art implementations Emma[Alexandrov et al., 2016] and DIQL [Fegaras and Noor, 2018], which are Domain Specific Languages (more detail about them in Section 3.5). The authors of Emma showed that their approach outperforms earlier works in [Alexandrov et al., 2016]. DIQL is a DSL built on monoid algebra (of which the μ-monoids algebra is an extension). Comparing against DIQL shows the interest of having a first-class fixpoint operator in the monoid algebra.

The expressions considered in these experiments are the ones presented in the examples (Section 4.2.5). The programs generated by μ-monoids from these expressions were obtained by systematically applying the rules $\mathrm{PF}, \mathrm{PJ}, \mathrm{PA}, \mathrm{P}_{\mathrm{dist}}$ (of Section 4.3). We evaluate these programs by comparing their execution times against the following programs:

- **DIQL**: The examples have been expressed using DIQL [Fegaras and Noor, 2018] queries. In particular, the fixpoint operation is expressed in terms of the more generic repeat operator of the DIQL language. We have written the queries in such a way that they compute the fixpoint more efficiently using the algorithm mentioned in 4.2.6.

- **Emma**: We used the example provided by Emma authors [Markl, 2019] to compute the TC queries, and we wrote modified versions to compute the SP and the path planning examples.

- **mu-monoid-no-PA** μ-monoids without the application of PA to assess the impact of the PA rule on the SP and the path planning examples.

- **manual-spark**: Hand written Spark program. It uses a loop in the driver to compute the fixpoint. So it is equivalent to the μ-monoids program without the $\mathrm{P}_{\mathrm{dist}}$ optimization. We use it to assess the impact of this optimization.

Dataset	Edges	Nodes	TC size
rnd_10k_0.001	50,119	10,000	5,718,306
rnd_20k_0.001	199,871	20,000	81,732,096
rnd_30k_0.001	450,904	30,000	255,097,974
rnd_10k_0.005	249,791	10,000	39,113,982
rnd_40k_0.001	799,961	40,000	531,677,274
rnd_50k_0.001	1,250,922	50,000	906,630,823

Dataset	Edges	Nodes
Yago	62,643,951	42,832,856
Facebook	88,234	4,039
DBLP	1,049,866	317,080

Table 4.1: Synthetic and real graphs used in experiments.

In addition to the examples of section 4.2.5, we evaluate two variants of TC and SP: TC filter and SP filter, where we compute the paths starting from a subset of 2000 nodes randomly chosen in the graph.

Datasets. We use two kinds of datasets:

- Real world graphs of different sizes, presented in Table 4.1, including a knowledge graph (the Yago [for Informatics and University, 2019] dataset[6]), a social network graph (Facebook), and a scientific collaborations network (DBLP) taken from [Leskovec, 2019].
- Synthetic graphs shown in Table 4.1, generated using the Erdos Renyi algorithm that, given an integer n and a probability p, generates a graph of n vertices in which two vertices are connected by an edge with a probability p. rnd_n_p denotes such a synthetic graph, whereas rnd_n_p_W denotes a rnd_n_p graph with edges weighted randomly (between 0 and 5).

  Other synthetic graphs are:

  - flight_n_p: where edges are taken from rnd_n_p with random depart and arrival times and duration assigned to them.
  - c_n_p: serialized object RDD files representing paths between cities. It is also generated from rnd_n_p, each city has been assigned up to 10 random landmarks.
  - u_n: serialized object RDD files of n users, each assigned up to 15 random movies.

Experimental setup. Experiments have been conducted on a Spark cluster composed of 5 machines (hence using 5 workers, one on each machine, and the driver on one of them)[7].

[6]We use a cleaned version of the real world dataset Yago 2s [for Informatics and University, 2019], that we have preprocessed in order to remove duplicate RDF [Cyganiak et al., 2014] triples (of the form <source, label, target>) and keep only triples with existing and valid identifiers. After preprocessing, we obtain a table of Yago facts with 83 predicates and 62,643,951 rows (graph edges).

[7]Each machine has 40 GB of RAM, 2 Intel Xeon E5-2630 v4 CPUs (2.20 GHz, 20 cores each) and 66 TB of 7200 RPM hard disk drives, running Spark 2.2.3 and Hadoop 2.8.4 inside Debian-based Docker containers.

For the Yago dataset, transitive closures are computed for the `isLocatedIn` edge label. The hand written spark program (manual-spark) has the optimizations PF and PA whenever possible, P_{dist} is the only rule it does not have. We have also written the DIQL queries in such a way they apply PF. Such a pre-filtering was not possible for Emma because the programs perform a non linear fixpoint. Trying to write a linear version leads to an exception in the execution. We were not able to write an Emma program that computes movie recommendations. Iterating over a users own movies leads to an exception.

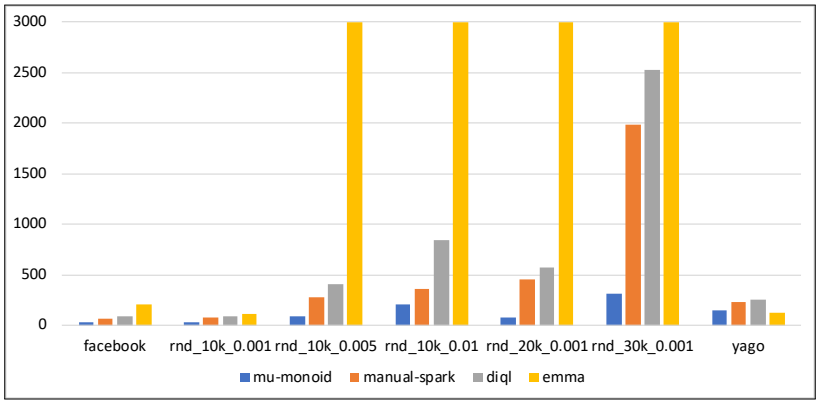


Figure 4.5: TC running times.

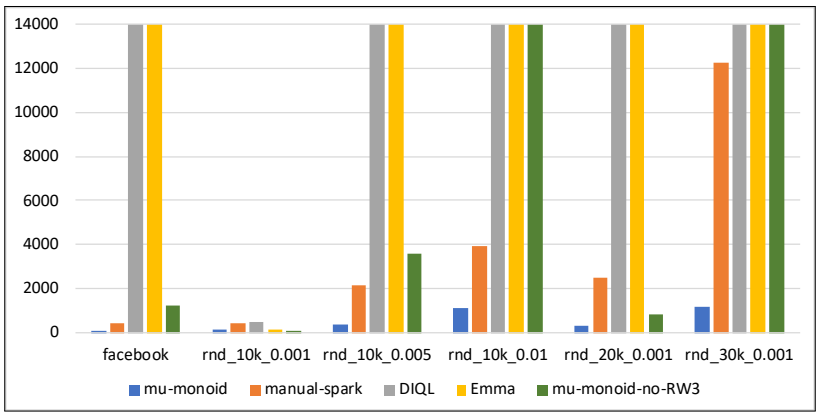


Figure 4.6: SP running times.

Results summary. Figure 4.10 presents the obtained results. We observe that the programs generated by μ-monoids systematically outperform the other program versions. The speedup is even more important for programs where PA is applied (SP, SP filter and path planning), especially when combined with P_{dist}.

This experimental comparison shows the benefit of the plan that distributes the fixpoint. It also highlights the benefits of the approach that synthesises code: generating programs that are not natural for a programmer to write, like the distributed loop to compute the fixpoint.

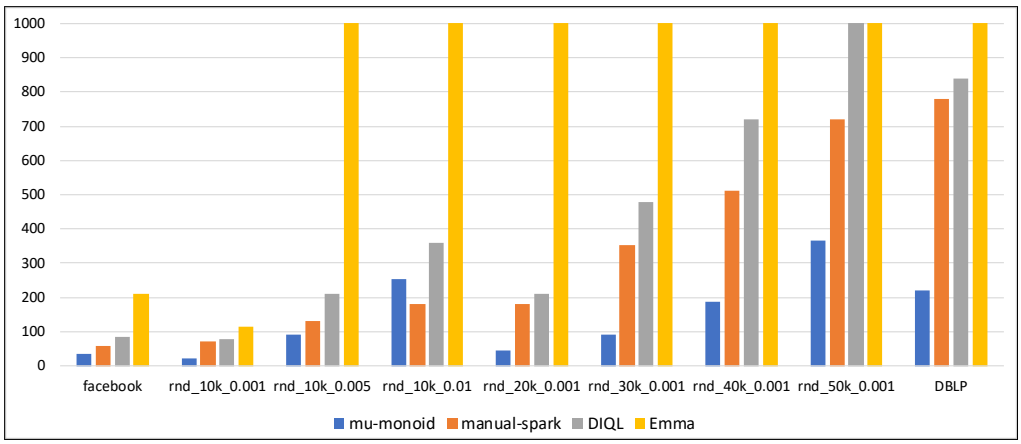


Figure 4.7: TC filter running times.

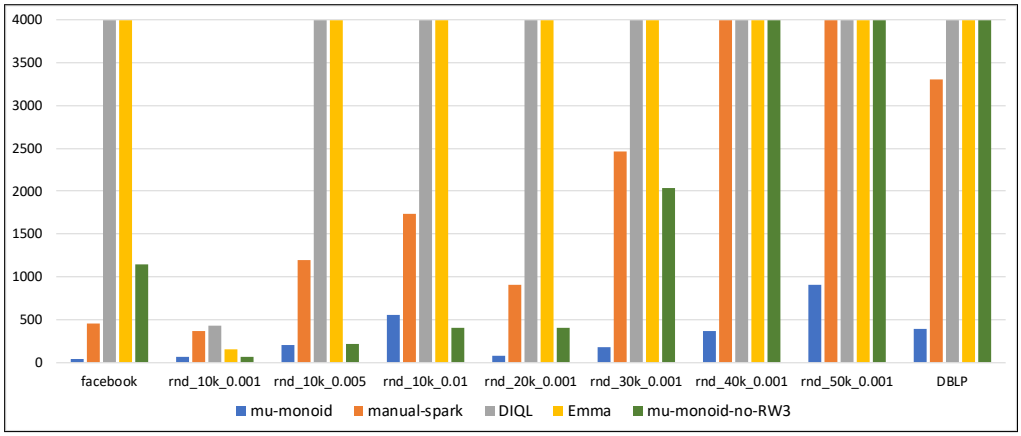


Figure 4.8: SP filter running times.

4.5 Conclusion

We propose to extend the monoid algebra with a fixpoint operator that models recursion. The extended μ-monoids algebra is suitable for modeling recursive computations with distributed data collections such as the ones found in big data frameworks. The major interest of the "μ" fixpoint operator is that, under prerequisites that are often met in practice, it can be considered as a monoid homomorphism and thus can be evaluated by parallel loops with one final merge rather than by a global loop requiring network overhead after each iteration.

We also propose rewriting rules for optimizing fixpoint terms: we show when and how filters can be pushed into fixpoints. In particular, we find a sufficient condition on the repeatedly evaluated term (φ) regardless of its shape, and we present a method using polymorphic types and a type system such as Scala's to check whether this condition holds. We also propose a rule to prefilter a fixpoint before a join. The third rule allows for pushing aggregation functions inside a fixpoint.

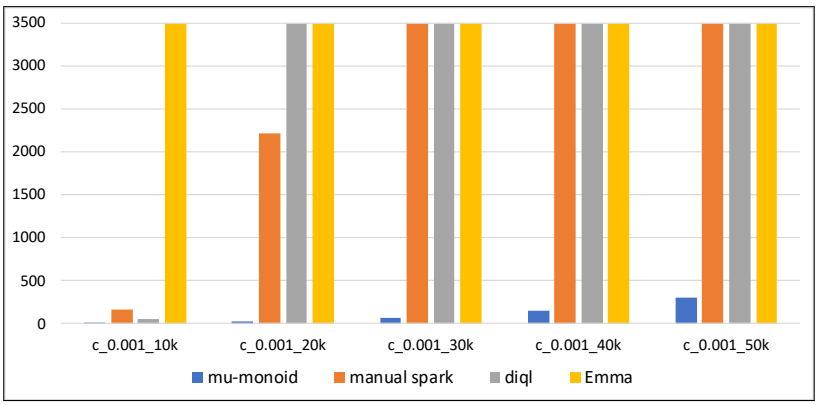


Figure 4.9: Path planning running times.

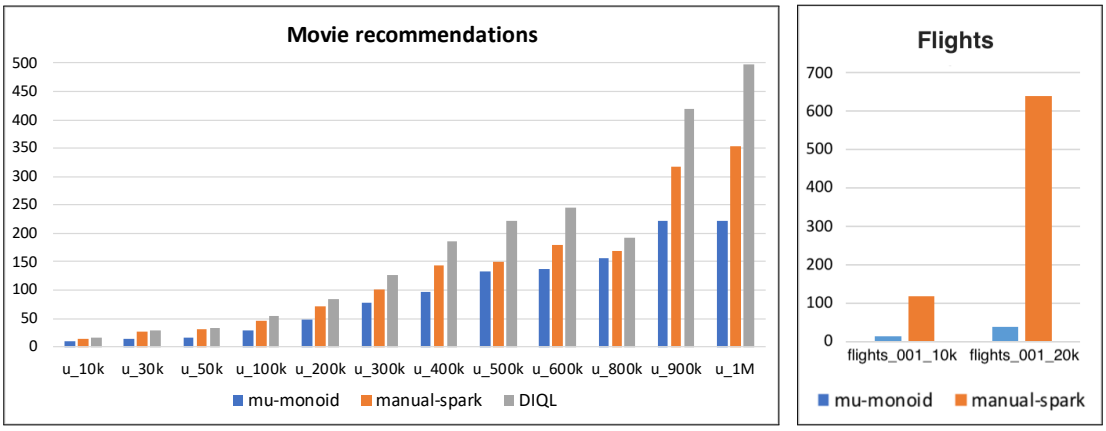


Figure 4.10: Movie recommendations and Flights running times.

Experiments suggest that: (i) Spark programs generated by the systematic application of these optimizations can be radically different from – and less intuitive – than the input ones written by the programmer; (ii) generated programs can be significantly more efficient. This illustrates the interest of developing optimizing compilers for programming with big data frameworks.

Chapter 5

General conclusion and perspectives

5.1 Conclusion

In this manuscript, we have studied the problem of evaluating queries on large datasets. We have considered the following aspects of this problem: performance and automatic optimization, large-scale data processing and distribution, expressivity, graph processing, reducing impedance mismatch, querying complex data, with a particular focus on the ability to express recursion. For this purpose, we have studied distributed evaluation of recursive queries using two formal approaches: RA and monoid algebra, which both can be used to express computations on distributed datasets. The first is a well studied and established framework for querying data, multiple works have tackled the question of optimizing relational queries, and the recent work on μ-RA studied the addition and optimization of recursion in RA but is limited to the centralized setting. We proposed Dist-μ-RA, a system that combines various techniques for the optimization and distribution of recursive relational queries. Regarding the algebraic aspect, it integrates well with the relational algebra and inherits its advantages including the fact that queries are optimized regardless of their initial shape and translation into the algebra. With respect to distribution, different strategies for evaluating recursive algebraic terms in a distributed setting have been studied. These strategies are implemented as plans with automated techniques for distributing data in order to reduce communication costs. Experimental results on both real and synthetic graphs show the effectiveness of the proposed approach compared to existing systems.

Despite all its advantages, the relational model is less flexible and leads to impedance mismatch problems as developped in Sections 1.1 and 2.1. Monoid algebra, the second formal approach we investigate, offers a framework for reasoning about distributed programs: monoids and monoid homomorphisms are well suited to model distributed computations. The evaluation of a monoid homomorphism can be broken down to the application of an associative operator. This associativity implies that parts of the computation can actually be performed in parallel and combined to get the final result. Since the data model consists of generic collections and the operations can be second order operations with arbitrary UDF, the formalism addresses the issues of relational algebra but is not as well established as relational algebra. We have proposed to extend monoid algebra with a fixpoint operator that models recursion. The extended μ-monoids algebra is suitable for modelling recursive computations with distributed data collections such as the ones found in Big Data frameworks. The major interest of the "μ" fixpoint operator is that, under prerequisites that are often met in practice, it can be considered as a monoid homomorphism and thus can be evaluated by parallel loops with one final merge rather than by a global loop requiring network overhead after each iteration. We also show how this algebra enables new optimizations through rewrite rules that push filters in fixpoints, prefilter fixpoints before joins, and push aggregations inside fixpoints. Experiments with the Spark platform illustrate performance gains brought by these systematic optimizations.

5.2 Perspectives

In order to provide users with effective tools to query large amounts of data, it is worthwhile to investigate ways to improve every level of the compilation process: from the DSL down to the generation of the physical plans that distribute the computations.

Regarding DSLs, one perspective is to investigate high-level languages that compile into the studied algebras, especially internal DSLs that allow for expressing recursion. Another perspective is to investigate the translation of Spark programs (or subsets of such programs) into μ-monoids so as to allow for their automatic optimization.

A DSL relies on a logical data model (the logical structure of the data, such as graphs, that the user queries using the DSL). A perspective would be to investigate how to translate this logical data model into appropriate relations (in the RA formalism) or collections (in the monoid formalism) depending on the query and the framework on top of which the physical plan of the query is executed. For instance, given a query, is it better to translate the queried graph into a collection of nodes and their neighbors or a collection of edges?

At the algebraic level, a future work would be to define normalization rules for μ-monoids expressions with the fixpoint operator in order to transform them to a form that is recognized by optimization rules. Another idea for future work is to investigate using the μ-monoids filter pushing technique to push filters into μ-monoids expressions other than the fixpoint.

One perspective to improve the execution of a recursive query is to perform dynamic changes to its generated physical plan in order to adapt to the load of data during fixpoint computations. For instance, changing the number of partitions or changing the type of a join from a broadcast join to a regular join depending on the size of the recursive variable at a given iteration.

So far, we have studied the execution of the physical plans of the queries on distributed frameworks like Spark. These frameworks are designed and optimized for distributed applications where data does not fit into the memory of a single machine and where scalability can be achived by adding more commodity hardware. A perspective would be to investigate the ability of the studied algebras to express and optimize applications that rather target highly parallel computing frameworks like MPI as well as to investigate their execution on these frameworks.

Bibliography

[gir, 2019] (2019). Apache giraph.

[gre, 2022] (2022). Apache tinkerpop. tinkerpop documentation.

[bdl, 2022] (2022). Bigdatalog repository.

[ico, 2022] (2022). The colorado index of complex networks (icon).

[dis, 2022] (2022). Dist-μ-RA system implementation.

[wik, 2022] (2022). Wikidata the free knowledge base.

[Abiteboul and Beeri, 1995] Abiteboul, S. and Beeri, C. (1995). The power of languages for the manipulation of complex values. *The VLDB Journal*, 4(4):727–794.

[Abiteboul et al., 1995] Abiteboul, S., Hull, R., and Vianu, V., editors (1995). *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition.

[Abul-Basher et al., 2017] Abul-Basher, Z., Yakovets, N., Godfrey, P., Ghajar-Khosravi, S., and Chignell, M. H. (2017). TASWEET: Optimizing Disjunctive Path Queries in Graph Databases. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, pages 470–473. OpenProceedings.org.

[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.

[Alexandrov et al., 2016] Alexandrov, A., Katsifodimos, A., Krastev, G., and Markl, V. (2016). Implicit parallelism through deep language embedding. *SIGMOD Record*, 45(1):51–58.

[Alexandrov et al., 2019] Alexandrov, A., Krastev, G., and Markl, V. (2019). Representations and optimizations for embedded parallel dataflow languages. *ACM Trans. Database Syst.*, 44(1):4:1–4:44.

[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).

[Angles and Gutierrez, 2018] Angles, R. and Gutierrez, C. (2018). *An Introduction to Graph Data Management: Fundamental Issues and Recent Developments*, pages 1–32.

[Arenas et al., 2021] Arenas, M., Gutierrez, C., and Sequeda, J. F. (2021). *Querying in the Age of Graph Databases and Knowledge Graphs*, page 2821–2828. Association for Computing Machinery, New York, NY, USA.

[Armbrust et al., 2015a] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. (2015a). Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA. Association for Computing Machinery.

[Armbrust et al., 2015b] Armbrust, M., Xin, R. S., Lian, C., Huai, Y., Liu, D., Bradley, J. K., Meng, X., Kaftan, T., Franklin, M. J., Ghodsi, A., and Zaharia, M. (2015b). Spark SQL: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1383–1394.

[Bagan et al., 2017] Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G. H. L., Lemay, A., and Advokaat, N. (2017). gmark: Schema-driven generation of graphs and queries. *IEEE Trans. Knowl. Data Eng.*, 29(4):856–869.

[Bancilhon et al., 1986] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. (1986). 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, pages 1–15, New York, NY, USA. ACM.

[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.

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

[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):31:1–31:46.

[Barceló et al., 2012] Barceló, P., Pérez, J., and Reutter, J. L. (2012). Relative expressiveness of nested regular expressions. In *AMW*.

[Bienvenu et al., 2014] Bienvenu, M., Calvanese, D., Ortiz, M., and Šimkus, M. (2014). Nested regular path queries in description logics. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'14, page 218–227. AAAI Press.

[Bonifati et al., 2018] Bonifati, A., Fletcher, G., Voigt, H., and Yakovets, N. (2018). *Querying Graphs*, volume 10 of *Synthesis Lectures on Data Management*.

[Bu et al., 2010] Bu, Y., Howe, B., Balazinska, M., and Ernst, M. D. (2010). Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1–2):285–296.

[Bunkenburg, 1993] Bunkenburg, A. (1993). The boom hierarchy. In *Functional Programming*.

[Carbone et al., 2015] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache flink^{TM}: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38.

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the 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, pages 404–416, New York, NY, USA. ACM.

[Cruz et al., 1987] Cruz, I. F., Mendelzon, A. O., and Wood, P. T. (1987). A graphical query language supporting recursion. *SIGMOD Rec.*, 16(3):323–330.

[Cyganiak et al., 2014] Cyganiak, R., Wood, D., and Lanthaler, M. (2014). Rdf 1.1 concepts and abstract syntax.

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 137–150.

[Ekanayake et al., 2010] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. (2010). Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, page 810–818, New York, NY, USA. Association for Computing Machinery.

[Fegaras, 2017] Fegaras, L. (2017). An algebra for distributed big data analytics. *Journal of Functional Programming*, 27:e27.

[Fegaras and Noor, 2018] Fegaras, L. and Noor, M. H. (2018). Compile-time code generation for embedded data-intensive query languages. In *2018 IEEE International Congress on Big Data, BigData Congress 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 1–8.

[Fire and Elovici, 2015] Fire, M. and Elovici, Y. (2015). Data mining of online genealogy datasets for revealing lifespan patterns in human population. *ACM Trans. Intell. Syst. Technol.*, 6(2).

[for Informatics and University, 2019] for Informatics, M. P. I. and University, T. P. (2019). YAGO: A high-quality knowledge base.

[Francis et al., 2018] Francis, N., Green, A., Guagliardo, P., Libkin, L., Lindaaker, T., Marsault, V., Plantikow, S., Rydberg, M., Selmer, P., and Taylor, A. (2018). Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1433–1445, New York, NY, USA. ACM.

[Gane et al., 2014] Gane, P., Bateman, A., Mj, M., O'Donovan, C., Magrane, M., Apweiler, R., Alpi, E., Antunes, R., Arganiska, J., Bely, B., Bingley, M., Bonilla, C., Britto, R., Bursteinas, B., Chavali, G., Cibrián-Uhalte, E., Ad, S., De Giorgi, M., Dogan, T., and Zhang, J. (2014). Uniprot: A hub for protein information. *Nucleic Acids Research*, 43:D204–D212.

[Giorgidze et al., 2011] Giorgidze, G., Grust, T., Schreiber, T., and Weijers, J. (2011). Haskell boards the ferry. volume 6647, pages 1–18.

[Gonzalez et al., 2014] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., and Stoica, I. (2014). Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 599–613.

[Graham Klyne, 2014] Graham Klyne, Jeremy J. Carroll, B. M. (2014). Rdf 1.1 concepts and abstract syntax.

[Green et al., 2013] Green, T. J., Huang, S. S., Loo, B. T., and Zhou, W. (2013). Datalog and recursive query processing. *Found. Trends Databases*, 5:105–195.

[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*, GRADES '13, pages 14:1–14:7, New York, NY, USA. ACM.

[Halliwell et al., 2021] Halliwell, N., Gandon, F., and Lecue, F. (2021). User Scored Evaluation of Non-Unique Explanations for Relational Graph Convolutional Network Link Prediction on Knowledge Graphs. In *International Conference on Knowledge Capture*, Virtual Event, United States.

[Harris and Seaborne, 2013] Harris, S. and Seaborne, A. (2013). SPARQL 1.1 Query Language, W3C recommendation.

[Ioannidis, 1986] Ioannidis, Y. E. (1986). On the computation of the transitive closure of relational operators. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, page 403–411, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Isard et al., 2007] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. (2007). Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 59–72, New York, NY, USA. Association for Computing Machinery.

[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. ACM.

[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.

[Kunegis, 2013] Kunegis, J. (2013). Konect: the koblenz network collection. pages 1343–1350.

[Lawal et al., 2020] Lawal, M., Genevès, P., and Layaïda, N. (2020). A cost estimation technique for recursive relational algebra. In d'Aquin, M., Dietze, S., Hauff, C., Curry, E., and Cudré-Mauroux, P., editors, *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, pages 3297–3300. ACM.

[Leskovec, 2019] Leskovec, J. (2019). Snap: Stanford large network dataset collection.

[Libkin et al., 2016] Libkin, L., Martens, W., and Vrgoč, D. (2016). Querying graphs with data. *J. ACM*, 63(2):14:1–14:53.

[Libkin and Vrgoč, 2012] Libkin, L. and Vrgoč, D. (2012). Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, page 74–85, New York, NY, USA. Association for Computing Machinery.

[Liénard et al., 2018] Liénard, J., Achakulvisut, T., Acuna, D., and David, S. (2018). Intellectual synthesis in mentorship determines success in academic careers. *Nature Communications*, 9.

[Malewicz et al., 2010] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G. (2010). Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International*

Conference on Management of Data, SIGMOD '10, page 135–146, New York, NY, USA. Association for Computing Machinery.

[Markl, 2019] Markl, V. (2019). Emma is a quotation-based scala dsl for scalable data analysis.

[Meijer et al., 2006] Meijer, E., Beckman, B., and Bierman, G. M. (2006). LINQ: reconciling object, relations and XML in the .net framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, page 706.

[Meijer and Bierman, 2011] Meijer, E. and Bierman, G. (2011). A co-relational model of data for large shared data banks. *Commun. ACM*, 54(4):49–58.

[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, pages 235–242, New York, NY, USA. ACM.

[Odersky et al., 2004] Odersky, M., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., Zenger, M., and et al. (2004). An overview of the scala programming language. Technical report.

[Olston et al., 2008] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 1099–1110, 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.

[Saccà and Zaniolo, 1986] Saccà, D. and Zaniolo, C. (1986). 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, pages 16–23, New York, NY, USA. ACM.

[Sakr et al., 2021] Sakr, S., Bonifati, A., Voigt, H., Iosup, A., Ammar, K., Angles, R., Aref, W., Arenas, M., Besta, M., Boncz, P. A., Daudjee, K., Valle, E. D., Dumbrava, S., Hartig, O., Haslhofer, B., Hegeman, T., Hidders, J., Hose, K., Iamnitchi, A., Kalavri, V., Kapp, H., Martens, W., Özsu, M. T., Peukert, E., Plantikow, S., Ragab, M., Ripeanu, M. R., Salihoglu, S., Schulz, C., Selmer, P., Sequeda, J. F., Shinavier, J., Szárnyas, G., Tommasini, R., Tumeo, A., Uta, A., Varbanescu, A. L., Wu, H.-Y., Yakovets, N., Yan, D., and Yoneki, E. (2021). The future is big graphs: A community view on graph processing systems. *Commun. ACM*, 64(9):62–71.

[Seib and Lausen, 1991] Seib, J. and Lausen, G. (1991). Parallelizing datalog programs by generalized pivoting. In *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '91, page 241–251, New York, NY, USA. Association for Computing Machinery.

[Seo et al., 2013] Seo, J., Park, J., Shin, J., and Lam, M. S. (2013). Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917.

[Sereni et al., 2008] Sereni, D., Avgustinov, P., and de Moor, O. (2008). Adding magic to an optimising datalog compiler. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 553–566, New York, NY, USA. Association for Computing Machinery.

[Seshadri et al., 1996] Seshadri, P., Hellerstein, J. M., Pirahesh, H., Leung, T. Y. C., Ramakrishnan, R., Srivastava, D., Stuckey, P. J., and Sudarshan, S. (1996). Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 435–446, New York, NY, USA. Association for Computing Machinery.

[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 Özcan, F., Koutrika, G., and Madden, S., editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1135–1149. ACM.

[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*, pages 661–672. ACM.

[Thusoo et al., 2009] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629.

[Wadler, 1989] Wadler, P. (1989). Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, page 347–359, New York, NY, USA. Association for Computing Machinery.

[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.

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

[Zaharia et al., 2016] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. (2016). Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65.

[Zaniolo et al., 2017] Zaniolo, C., Yang, M., Das, A., Shkapsky, A., Condie, T., and Interlandi, M. (2017). Fixpoint semantics and optimization of recursive

datalog programs with aggregates. *Theory Pract. Log. Program.*, 17(5-6):1048–1065.