Efficient Iterative Programs with Distributed Data Collections

Sarah Chlyah, Nils Gesbert, Pierre Genevès, Nabil Layaïda

EFFICIENT ITERATIVE PROGRAMS WITH DISTRIBUTED DATA COLLECTIONS

SARAH CHLYAH, NILS GESBERT, PIERRE GENEVÈS, AND NABIL LAYAÏDA

Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble
e-mail address: sarah.chlyah@inria.fr, nils.gesbert@inria.fr, pierre.geneves@inria.fr, nabil.layaida@inria.fr

Abstract. Big data programming frameworks have become increasingly important for the development of applications for which performance and scalability are critical. In those complex frameworks, optimizing code by hand is hard and time-consuming, making automated optimization particularly necessary. In order to automate optimization, a prerequisite is to find suitable abstractions to represent programs; for instance, algebras based on monads or monoids to represent distributed data collections. Currently, however, such algebras do not represent recursive programs in a way which allows for analyzing or rewriting them. In this paper, we extend a monoid algebra with a fixpoint operator for representing recursion as a first class citizen and show how it enables new optimizations. Experiments with the Spark platform illustrate performance gains brought by these systematic optimizations.

1. 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 [DG04], Dryad [IBY+07], Spark [ZXW+16], Flink [CKE+15]. While these systems can handle large amounts of data and allow users to write a broad range of applications, writing efficient applications 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 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 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

Key words and phrases: fixpoint operator, distributed data, rewrite rules, optimization.

Preprint submitted to
Logical Methods in Computer Science

```
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) })
    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 1. Shortest paths program.

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 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.

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.

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 in order to express complex computations on more complex data like nested collections, using a formalism based on the relational model requires flattening the data and using ad-hoc solutions for supporting user defined functions (UDFs). This means

```
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 2. Shortest paths program with less data exchange.

that: (1) At the language level, we could have a query language expressed on a flat data model which causes *impedance mismatch* issues. It is the term that is used to refer to the problems that arise when the data model of the high level language is different from that of the general purpose host language. Specifically, 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. In addition, the DSL provides limited support for complex data processing (data transformation, iteration, aggregation, etc.). In order to perform custom transformations on data, 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. (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 that manipulate data in their native format. As argued by Meijer in [MB11], 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. 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.). 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 [RRV17, LV12, JGGL20]. 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 from how it is executed.

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 [FN18, Feg17] 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).

Contributions. Our contributions are the following:

(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;

[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.

- 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.

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. It is based upon the monoid algebra of Fegaras [Feg17]. 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.

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. 2.2 we recall the main definitions of the monoid algebra proposed by Fegaras [Feg17]. We then introduce a general notion of *aggregation function* in Sec. 2.3 and our addition to the monoid algebra, the fixpoint operator μ, in Sec. 2.4. Then in Sec. 2.5 we define the syntax of our own core calculus, and, in Sec. 2.6, a minimal type system for it. We then proceed to give a denotational semantics for our specific constructs in Sec. 2.7 and discuss evaluation of expressions in Sec. 2.9.

2.1. **Data model: distributed collections of data.**

2.1.1. *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

[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.

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

Let us now recall the algebraic definition of a *monoid*:

Definition 2.1. A *monoid* is a triple $(S, \otimes, e)$ where S is a set, $\otimes$ an associative binary operation on S, and e a neutral element for $\otimes$, i. e. such that:

$$\forall x, y, z \in S \quad x \otimes (y \otimes z) = (x \otimes y) \otimes z$$
$$\forall x \in S \quad x \otimes e = x = e \otimes x$$

As noticed by Fegaras [Feg17], our three 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. To formalize this, we recall the algebraic definitions of congruence and quotient monoid:

Definition 2.2 Congruence. Let $(A, \otimes, e)$ be a monoid. Let $\sim$ be an equivalence relation on A. We say that $\sim$ is a *congruence* on the monoid if it is *compatible* with $\otimes$, i. e. if:

$$\forall a, b, a', b' \quad a \sim a' \wedge b \sim b' \Rightarrow a \otimes b \sim a' \otimes b'$$

Definition 2.3 Quotient monoid. Let $(A, \otimes, e)$ be a monoid and $\sim$ a congruence on it. For $a \in A$, let $\hat{a} = \{b \in A \mid b \sim a\}$, the equivalence class of a. Let $\hat{A} = \{\hat{a} \mid a \in A\}$ be the set of all equivalence classes, and let $\hat{\otimes}$ be the operation on $\hat{A}$ defined by $\hat{a} \mathbin{\hat{\otimes}} \hat{b} \stackrel{\text{def}}{=} \widehat{a \otimes b}$. This is well-defined because $\sim$ is compatible with $\otimes$, so that the result is the same independently of the particular choice of a and b in their equivalence class.

Then $(\hat{A}, \hat{\otimes}, \hat{e})$ is a monoid, termed the *quotient monoid* of $(A, \otimes, e)$ by $\sim$ and noted $(A, \otimes, e)/\sim$.

Let $\sim_{\text{comm}}$ be the congruence on $(\mathsf{List}[t], +\!+, [\,])$ generated by commutativity, i. e. the smallest congruence such that: $\forall a, b \; a +\!+ b \sim_{\text{comm}} b +\!+ a$. 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 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.

2.1.2. *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:

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

As mentioned previously (Sec. 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:

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

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.

2.2. **Fegaras' monoid algebra.** In this section, we recall briefly the main definitions from Fegaras' monoid algebra [Feg17], upon which our work is based.

The monoid algebra is based on the three sorts of collection monoids described in 2.1.1 and on *collection homomorphisms*.

We first recall the definition of a monoid homomorphism:

Definition 2.4. Let $(A, \otimes, e)$ and $(B, \odot, \varepsilon)$ be two monoids. A *monoid homomorphism* from A to B is a function h from A to B such that:

$$h(e) = \varepsilon \text{ and}$$
$$\forall x, y \in A \quad h(x \otimes y) = h(x) \odot (y).$$

Collection homomorphisms are now defined using the following universal property enjoyed by the collection monoids (which are *free structures* in the algebraic sense):

Property 1 universal property of collection monoids. *For Coll a collection monoid, let $\mathbb{U}_{Coll}$ be the corresponding singleton construction function. Let $(A, \otimes, e)$ be a monoid which satisfies all the algebraic laws of Coll (i. e. commutativity for bags, and commutativity and idempotence for sets). Let $f : t \to A$ be a function.*

Then there exists a unique monoid homomorphism $H^{\otimes}_f$: Coll[t] $\to$ A such that: $H^{\otimes}_f(\mathbb{U}_{Coll}(x)) = f(x)$.

This *collection homomorphism* applies the function f to all the elements of the input collection and combines all the results together with the operation $\otimes$, yielding a single element of A.

Fegaras' monoid algebra comprises a number of collection homomorphisms, all defined in the form $H^{\otimes}_f$ for appropriate fs and $\otimes$s. We refer the reader to [Feg17] for the detail. In the present work, we use a slightly different set. Namely:

- we do not consider orderBy because we concentrate on bags rather than lists;
- we use reduceByKey rather than groupBy (together with the other operations, they lead to the same expressivity);
- we add the join operation. Even though it can be expressed in terms of coGroup, this operation is useful for us to have as a primitive because it is an homomorphism from each of its two arguments separately, whereas coGroup is only a binary homomorphism (an homomorphism from the product monoid of its two arguments).

Our set of primitive operations is thus the following, here presented in a way adapted to our default abstraction level of bags:

- $\mathsf{flmap}(f, X)$, 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[3], 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 a monoid homomorphism 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

[3]Commutativity is needed because the monoid of bags is commutative (see Proposition 1)

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. This operation is an homomorphism from the product monoid $\mathsf{Bag}[t_1 \times t_2] \times \mathsf{Bag}[t_1 \times t_3]$ to $\mathsf{Bag}[t_1 \times (\mathsf{Bag}_l[t_2] \times \mathsf{Bag}_l[t_3])]$.

Additionnally, Fegaras' monoid algebra includes a repeat operation, which is not an homomorphism. In this work, we replace this operation with our own μ operation, explained in detail in Section 2.4. Before we get there, we introduce our notion of *aggregation function*.

2.3. **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. 2.1.1). 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 distinct, as we will detail below. We regroup such functions under the term *aggregation functions*.

Definition 2.5 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 2.6. The equivalence relation associated with an aggregation function, $\sim_\delta$, is defined by:

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

Lemma 2.7. *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 2.8. Let δ be an aggregation function, we define the binary operation $\otimes_\delta$ on bags as follows:

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

Lemma 2.9. *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 2.10. 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 $\sim_{\text{idem}}$; $\otimes_{\mathsf{distinct}}$ 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 2.11. *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 following three 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 2.12. 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 [ZYD+17], as made more apparent by property (3).

Property 2. *$\mathsf{distinct}$ 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, $\mathsf{distinct}(\varphi(A)) = \mathsf{distinct}(\varphi(\mathsf{distinct}(A)))$. □

2.4. **The μ operation.** Our purpose is to extend the monoid algebra (as defined in 2.2) 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 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 2.13 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. 2.12). 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 $\mathsf{distinct}$.

In the following, we consider $\mathsf{distinct}$ 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. 3.3).

2.5. **Syntax of μ-monoids.** 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
	$\mid$	$\lambda\ \langle \pi_1 \to e_1 \mid \cdots \mid \pi_n \to e_n \rangle$	function with pattern matching
	$\mid$	$e\ e \mid C(e_1, e_2, ..., e_n)$	application, constructor expression
	$\mid$	$\mathsf{flmap}(e, e) \mid \mathsf{reduce}(e, e, e)$	flatmap, reduce
	$\mid$	$\mathsf{reduceByKey}(e, e) \mid \mathsf{cogroup}(e, e) \mid \mathsf{join}(e, e)$	reduce by key, cogroup, join by key
	$\mid$	$\mu_e\,(e, e)$	fixpoint

To this, we add the following as syntactic sugar:

- $(e_1, ..., e_n)$ with no constructor is an abbreviation for: $\mathsf{Tuple}(e_1, ..., e_n)$
- $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ 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 $\mathsf{let}\ \mathbf{name} = e_1\ \mathsf{in}\ e_2$ syntax with the usual meaning.

Example:

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

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 $\mathsf{contains}$ are defined in $\mathcal{L}$: $+$ appends its second argument to the first and $\mathsf{contains}$ checks whether the first argument is contained in the second argument. $\mathsf{appendToWords}$ 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:

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

2.6. **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 2.14 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 2.15. 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 2.16 Subtyping. We define subtyping as follows (it is nontrivial only for sum types):

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

Definition 2.17 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 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 $\mathcal{L}$.

2.6.1. *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

$$\frac{\Gamma \vdash e_1 : t \to \mathsf{Bag}_l[t'] \qquad \Gamma \vdash e_2 : T[t]}{\Gamma \vdash \mathsf{flmap}(e_1, e_2) : T[t']} \qquad \frac{\Gamma \vdash e_1 : t \to t \to t \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : T[t]}{\Gamma \vdash \mathsf{reduce}(e_1, e_2, e_3) : t}$$

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

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

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

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

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

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

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

FIGURE 3. Typing judgements.

identify a subset of functions that can be statically checked. It is the set of terms φ of the form $\lambda\ \langle X \to \mathrm{H}(X) \rangle$ where $\mathrm{H}(X)$ is defined as follows:

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

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[4]. In our case, 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.

For example, the program shown in Figure 1 can be expressed as a fixpoint $\mu_\delta\,(R, \varphi)$ where φ is an homomorphism of the form $\lambda\ \langle X \to \mathrm{H}(X) \rangle$ which is in charge of computing new paths and their lengths. We will further detail this example after having described the denotational semantics of the algebraic operators involved.

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

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

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

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

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

$$\mu_\delta\,(R, \varphi) = \begin{cases} S_N & \text{if there exists N such that } S_{N+1} = S_N \\ \omega & \text{otherwise} \end{cases} \quad \text{where } S_n = \bigotimes_{0 \le k \le n}^{\delta} (\delta \circ \varphi)^k(\delta(R))$$

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

FIGURE 4. Denotational semantics

2.7. **μ-monoids denotational semantics.** Figure 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. 2.6.1. 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 [Feg17], as discussed in 2.4. We can check that they are still monoid homomorphisms if we add ω to all the monoids as an absorbing element[5].

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:

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

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 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[6] of Ψ.

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

Theorem 2.18. *$\mu_\delta\,(\cdot, \varphi)$ 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 $S_n^i = \bigotimes_{k \le n}^{\delta} (\delta \circ \varphi)^k(\delta(R_i))$. 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

[5]For a monoid $(S, \otimes, e)$, an absorbing element ω satisfies the following $\forall x \in S x \otimes \omega = \omega = \omega \times x$.

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

some point, say N_1 and N_2, then (S_n^0) becomes stationary at $max(N_1, N_2)$ and we do have $\mu_\delta(R_0, \varphi) = \mu_\delta(R_1, \varphi) \otimes_\delta \mu_\delta(R_2, \varphi)$. □

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

Transitive closure (TC).

$$\begin{aligned}&\mathsf{let}\ \mathbf{reverse\_edges} = \lambda\ \langle (a,b) \to \{\!\{(b,a)\}\!\}\rangle\ \mathsf{in}\\ &\mathsf{let}\ \mathbf{drop\_mid} = \lambda\ \langle (mid,(src,dest)) \to \{\!\{(src,dest)\}\!\}\rangle\ \mathsf{in}\\ &\mu(R, \lambda\ \langle X \to \mathsf{flmap}(\mathbf{drop\_mid}, \mathsf{join}(\mathsf{flmap}(\mathbf{reverse\_edges}, X), R))\rangle)\end{aligned}$$

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 $\mathsf{join}(\mathsf{flmap}(\mathbf{reverse\_edges}, X), R)$ 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. This intermediary node is the join key, thus the join constructs pairs of the form $(mid, (src, dest))$ where mid is the intermediary node, which must then be dropped by another flatmap. 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).

$$\begin{aligned}&\mathsf{let}\ \mathbf{key\_dest} = \lambda\ \langle ((src,dest),w) \to \{\!\{(dst,(src,w))\}\!\}\rangle\ \mathsf{in}\\ &\mathsf{let}\ \mathbf{key\_src} = \lambda\ \langle ((src,dest),w) \to \{\!\{(src,(dest,w))\}\!\}\rangle\ \mathsf{in}\\ &\mathsf{let}\ \mathbf{combine} = \lambda\ \langle (mid,((src,w_1),(dest,w_2))) \to \{\!\{((src,dest),w_1+w_2)\}\!\}\rangle\ \mathsf{in}\\ &\mathsf{let}\ \mathbf{all\_paths} = \mu(R, \lambda\ \langle X \to \mathsf{flmap}(\mathbf{combine}, \mathsf{join}(\mathsf{flmap}(\mathbf{key\_dest}, X), \mathsf{flmap}(\mathbf{key\_src}, R)))\rangle)\ \mathsf{in}\\ &\mathsf{reduceByKey}(\min, \mathbf{all\_paths})\end{aligned}$$

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 reduceByKey operation keeps the shortest paths between each pair of nodes.

Flights.

$$\begin{aligned}&\mathsf{let}\ \mathbf{corr\_possible} = \lambda\ \langle (\mathsf{corr}, (\mathsf{Flight}(\mathsf{dtime1}, \mathsf{atime1}, \mathsf{dep1}, \mathsf{dest1}, \mathsf{dur1}), \mathsf{Flight}(\mathsf{dtime2}, \mathsf{atime2}, \mathsf{dep2}, \mathsf{dest2}, \mathsf{dur2}))) \to\\ &\qquad \mathsf{if}\ \mathsf{atime1} < \mathsf{dtime2}\ \mathsf{then}\ \{\!\{\mathsf{Flight}(\mathsf{dtime1}, \mathsf{atime2}, \mathsf{dep1}, \mathsf{dest2}, \mathsf{dur1}+\mathsf{dur2})\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle\ \mathsf{in}\\ &\mathsf{let}\ \mathbf{key\_dest} = \lambda\ \langle \mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{dep}, \mathsf{dest}, \mathsf{dur}) \to \{\!\{(\mathsf{dest}, \mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{dep}, \mathsf{dest}, \mathsf{dur}))\}\!\}\rangle\ \mathsf{in}\\ &\mathsf{let}\ \mathbf{key\_dep} = \lambda\ \langle \mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{dep}, \mathsf{dest}, \mathsf{dur}) \to \{\!\{(\mathsf{dep}, \mathsf{Flight}(\mathsf{dtime}, \mathsf{atime}, \mathsf{dep}, \mathsf{dest}, \mathsf{dur}))\}\!\}\rangle\ \mathsf{in}\\ &\mu(R, \lambda\ \langle X \to \mathsf{flmap}(\mathbf{corr\_possible}, \mathsf{join}(\mathsf{flmap}(\mathbf{key\_dest}, X), \mathsf{flmap}(\mathbf{key\_dep}, R)))\rangle)\end{aligned}$$

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.

$$\begin{array}{l}
\mathsf{let}\ \mathbf{paths} = \lambda\ \langle(\mathsf{City}(n_1, l_1), \mathsf{City}(n_2, l_2)) \to \{\!\{(\mathsf{Path}(n_1, n_2), l_1 +\!\!+\, l_2)\}\!\}\rangle\ \mathsf{in} \\
\mathsf{let}\ \mathbf{key_name_dep} = \lambda\ \langle(\mathsf{Path}(s, d), l) \to \{\!\{(s, (\mathsf{Path}(s, d), l))\}\!\}\rangle\ \mathsf{in} \\
\mathsf{let}\ \mathbf{key_name_dest} = \lambda\ \langle(\mathsf{Path}(s, d), l) \to \{\!\{(d, (\mathsf{Path}(s, d), l))\}\!\}\rangle\ \mathsf{in} \\
\mathsf{let}\ \mathbf{combine} = \lambda\ \langle(k, ((\mathsf{Path}(s_1, d_1), l_1), (\mathsf{Path}(s_2, d_2), l_2))) \to \{\!\{(\mathsf{Path}(s_1, d_2), l_1 +\!\!+\, l_2)\}\!\}\rangle\ \mathsf{in} \\
\mathsf{let}\ \mathbf{all_paths} = \mu(\mathsf{flmap}(\mathbf{paths}, R), \lambda\ \langle X \to \mathsf{flmap}(\mathbf{combine}, \mathsf{join}(\mathsf{flmap}(\mathbf{key_name_dest}, X), \\
\qquad \mathsf{flmap}(\mathbf{key_name_dep}, \mathsf{flmap}(\mathbf{paths}, R)))))\rangle)\ \mathsf{in} \\
\mathsf{flmap}(\lambda\ \langle(\mathsf{Path}(s, d), l) \to \mathsf{if}\ s = \text{"Paris"}\ \text{and}\ d = \text{"Geneva"}\ \mathsf{then}\ \{\!\{(\mathsf{Path}(s, d), l)\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle, \\
\qquad \mathsf{reduceByKey}(\mathsf{bestRated}, \mathsf{flmap}(\lambda\ \langle(s, d, l) \to (\mathsf{Path}(s, d), l))\rangle, \mathbf{all_paths})))
\end{array}$$

where R is a set of routes between two cities, represented as pairs of cities. Each city $\mathsf{City}(n, l)$ has a name n and a list 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 **all_paths** 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.

$$\begin{array}{l}
\mathsf{let}\ \mathbf{users_who_like} = \lambda\ \langle x \to \mathsf{flmap}(\lambda\ \langle\mathrm{User}(u, bm) \to \mathsf{if}\ x \in bm\ \mathsf{then}\ bm\ \mathsf{else}\ \{\!\{\}\!\}\rangle, U)\rangle\ \mathsf{in} \\
\mu(S, \lambda\ \langle X \to \mathsf{flmap}(\mathbf{users_who_like}, X)\rangle)
\end{array}$$

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.

2.9. **Evaluation of expressions.**

2.9.1. *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:

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

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

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 $\perp$. Let i be the smallest index such that $\mathrm{m}(v, \pi_i) = S \neq \perp$, 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. 2.13. 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:

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

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

2.9.2. *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.

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. 3.1.1.
- 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 [ZYD+17].
- 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.

3.1. **Pushing filter inside a fixpoint (PF).**

3.1.1. *Filter depending on a single pattern variable.*

Definition 3.1 filter. We call *filter* a function of the form:
$$\lambda\,\langle D \to \mathsf{flmap}(\lambda\,\langle \pi \to \mathsf{if}\ c(a)\ \mathsf{then}\ \{\!\{\pi\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle, D)\rangle$$
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 3.2. *$\forall a \in \mu(R,\varphi) \quad a \in \varphi^{(n)}(\{\!\{r\}\!\})$ 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 [Wad89]: 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 [OMM+04] to get the output type[7]. $C(\alpha)$ 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.

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 3.3 path. We define the *path* to the node labelled n in the tree T denoted $\text{path}(n, T)$ by the ordered sequence $\text{Seq}(a_i)$ 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.

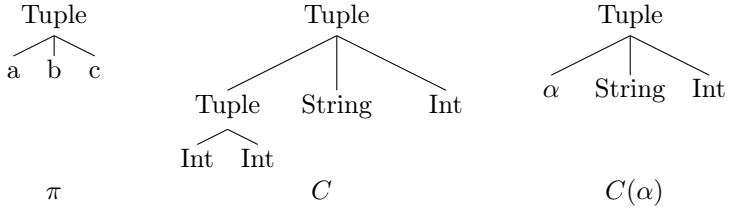
Let us consider the function $\texttt{replace}_\alpha(p, T)$ that, given a path p and a type T returns a polymorphic type $T(\alpha)$ that is obtained by replacing in T the node at path p and its children by a node labelled α. Let us now consider $C(\alpha) = replace_\alpha(path(a, \pi), C)$, where $\mathsf{Bag}[C]$ is the input type of φ. Note that this path makes sense in C because C matches π (see Appendix 2.6).

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

$$e : C \text{ and } \pi_a(e) : \alpha \Rightarrow e : C(\alpha) \tag{3.1}$$

$$e : C(\alpha) \Rightarrow \pi_a(e) : \alpha \tag{3.2}$$

For example:



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

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).

3.1.2. *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 [FN18] to push filters in a $\mathsf{cogroup}$ or a $\mathsf{groupby}$). 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.

3.2. **Filtering inside a fixpoint before a join (PJ).** Let us consider the expression: $\mathsf{join}(A,\ B)$, where $B = \mu(R, \varphi)$. 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 $F_A(B) = \{(k, v) \mid (k, v) \in B \wedge \exists w\ (k, w) \in A\}$:

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

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 $F_A(B) = \mathsf{flmap}(\lambda\ \langle (k, v) \to \mathsf{if}\ c(k)\ \mathsf{then}\ \{\!\{(k, v)\}\!\}\ \mathsf{else}\ \{\!\{\}\!\}\rangle, B)$ where $c(k)$ is the boolean expression that corresponds to the predicate $\exists w (k, w) \in A$. This expression can be: $c(k) = \mathsf{reduce}(\vee, e_\vee, \mathsf{flmap}(\lambda\ \langle (k', a) \to k == k'\rangle, A))$. 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:

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

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 $(k,x) \in C$.

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 3.4. *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.2.8) 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. The requirement of compatibility of δ with φ 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. So, in practice, we require an annotation from the programmer on the aggregation operations that verify the necessary constraints. However, we can list common known aggregation functions : $\mathsf{reduceByKey}(f,\cdot)$, filters (see Def. 3.1), mainly.

3.4. **Distribution of the fixpoint operations ($\mathbf{P_{dist}}$).** As explained in 2.9.1, 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 $\mathcal{P}_1$. $\mathcal{P}_1$ 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. 2.8), 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 $\mathcal{P}_2$, 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 $\mathcal{P}_2$ instead of $\mathcal{P}_1$ for evaluating fixpoints.

3.4.1. *Avoiding $\cup^{nl}$ in $\mathcal{P}_2$.* In the common case where δ is distinct, $\mathcal{P}_2$ 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 3.5. *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 3.2. 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 3.1.1. We explore every node n in C (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.

3.5. **Effects of the rules on performance.**

In this section, we discuss the impacts of the rules and the conditions under which they produce terms that are more efficient in practice. The verification of these conditions is outside of the scope of this paper. Techniques that estimate the size of algebraic expressions such as those found in [LGL20] can be used to perform such verifications.

3.5.1. *PF effects.* 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.

3.5.2. *PJ effects.* The rule PJ introduces an additional cogroup to compute the filter being pushed in the fixpoint (as detailed in Sec. 3.2). The cost of evaluating a term depends on two important aspects: 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. In the worst case (the filter does not remove any result), the additional cogroup does not change the worst case complexity of the computation (join and cogroup have the same worst case complexity $O(n^2)$).

To analyse the impact of the rule on non-local data transfers, we need to estimate and compare the size of the 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 2.9.2, all our algebraic operators apart from flatmap trigger non-local transfers. Let $size(t)$ be the size of the result obtained by evaluating t, $size_t(t)$ the size of transfers incurred by the evaluation of t, and N the number of partitions (parallel tasks) in the cluster. We consider the following:

- Repartitioning a dataset A by key requires all A to be tranferred across the network (each element of A has to be in the partition corresponding to its key). This means that $\mathsf{size}_t(\mathsf{groupby}(t)) = size(t)$, and $\mathsf{size}_t(\mathsf{cogroup}(t_1, t_2)) = \mathsf{size}_t(\mathsf{join}(t_1, t_2)) = \mathsf{size}(t_1) + \mathsf{size}(t_2)$.
- $\mathsf{size}_t(\mathsf{distinct}(A)) = N \times \mathsf{size}(A)$ because all A has to be seen by each partition so that duplicates can be removed globally. This means that $\mathsf{size}_t(\mu(R, \varphi)) = N \times \mathsf{size}(\mu(R, \varphi))$.

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

$S_1 = \mathsf{size}(A) + (N + 1) \times \mathsf{size}(\mu(R, \varphi))$, 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.

$S_2 = 2 \times \mathsf{size}(A) + (N+1) \times \mathsf{size}(\mu(F_A(R), \varphi)) + \mathsf{size}(R)$, 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 need to compare S_1 and S_2, which amounts to comparing the following quantities: $(N+1) \times \mathsf{size}(\mu(R, \varphi))$ and $(N+1) \times \mathsf{size}(\mu(F_A(R), \varphi)) + \mathsf{size}(A) + \mathsf{size}(R)$. In other words, PJ improves data transfers when 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. This is likely to be the case since the data obtained at each iteration of the fixpoint (including R) is filtered.

3.5.3. *PA effects.* The PA rule applies the aggregation function δ on the fixpoint's intermediate results instead of once at the end. Whenever δ reduces the size of these results, 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.

3.5.4. *P_{dist} effects.* Application of $\mathrm{P}_{\mathrm{dist}}$ can drastically decrease data transfers across the network. As explained in Sec. 3.4, plan $\mathcal{P}_2$ avoids transferring intermediate results during fixpoint iterations or even entirely (if a data partioning that verifies the criteria presented in 3.4.1 exists).

The efficiency of the two plans that distribute the fixpoint depends on two aspects. First, for a term $\mu(R, \varphi)$ to be evaluated on a plateform like Spark, 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 $\mathcal{P}_2$: when those datasets become too large to be handled by one worker, $\mathcal{P}_1$ is more appropriate. Second, a factor that determines the efficiency of $\mathcal{P}_2$ (and impacts the size of the iteration results X) is the number of parallel tasks that execute the program. In Spark, this corresponds to the number of partitions. 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, estimation of an appropriate number of partitions for $\mathcal{P}_2$ would ideally be based on an estimated size of the constant part, the size of intermediate data produced by the fixpoint and the workers memory capacity. In the experiments we present below, we use a simple heuristic to determine the number of partitions: 4 times the total number of cores of the cluster.

4. Experimental results

Methodology. We experiment the μ-monoids approach in the context of the Spark platform [ZXW+16].

We evaluate Spark programs generated from optimized μ-monoids expressions, and compare their performance with the state-of-the-art implementations Emma[AKKM16] and DIQL [FN18], which are Domain Specific Languages (more detail about them in Section 5). The authors of Emma showed that their approach outperforms earlier works in [AKKM16]. 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.

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). Each machine has 128 Go of RAM and the Spark worker on this machine is configured to use 40 GB, 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.

Algorithms. The algorithms considered in these experiments are: TC, SP, Flights, Path Planning, and Movie Recommendations presented in the examples (Section 2.8). In addition, 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 input graph.

Systems. μ-monoids is evaluated against other systems on the algorithms mentioned above. μ-monoids programs are generated from the μ-monoids terms expressing these algorithms (see Examples Section 2.8) and by systematically applying the rules $\mathrm{PF}, \mathrm{PJ}, \mathrm{PA}, \mathrm{P_{dist}}$ (of Section 3). We evaluate these programs by comparing their execution times against the following programs:

- **DIQL**: The algorithms have been expressed using DIQL [FN18] 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 2.9.1. The DIQL system is available at [diq].
- **Emma**: We used the example provided by Emma authors [Mar19] to compute the TC queries, and we wrote modified versions to compute the SP and the path planning examples. The Emma system is available at [Mar19].
- **μ-monoids-no-PA**: μ-monoids without the application of PA to assess the impact of the PA rule.
- **μ-monoids-no-$\mathbf{P_{dist}}$**: μ-monoids without the application of $\mathrm{P_{dist}}$ to assess the impact of the $\mathrm{P_{dist}}$ rule.

All these systems run on the same experimental setup presented above and on the same Spark plateform.

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.

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_10k_0.01	499,486	10,000	45,098,336
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 1. Synthetic and real graphs used in experiments.

Datasets. We use two kinds of datasets:

- Real world graphs of different sizes, presented in Table 1, including a knowledge graph (the Yago [fIU19] dataset[8]), a social network graph (Facebook), and a scientific collaborations network (DBLP) taken from [Les19].
- Synthetic graphs shown in Table 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.

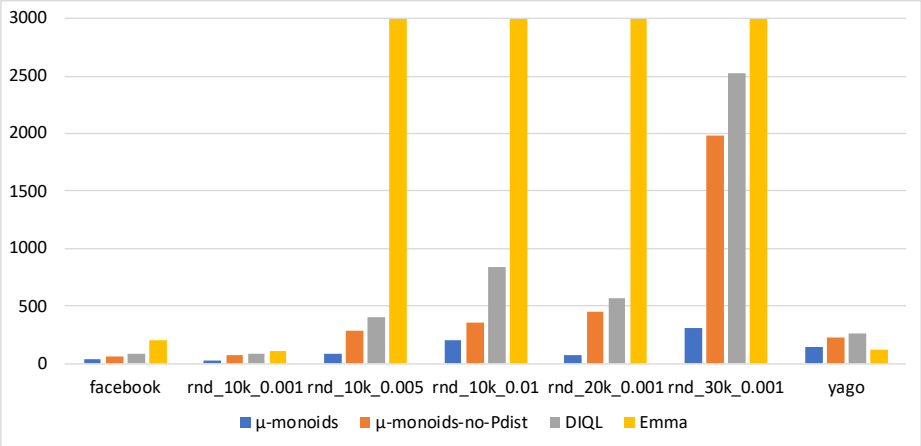


FIGURE 5. TC running times.

[8]We use a cleaned version of the real world dataset Yago 2s [fIU19], that we have preprocessed in order to remove duplicate RDF [CWL14] 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).

For this dataset, transitive closures are computed for the `isLocatedIn` edge label.

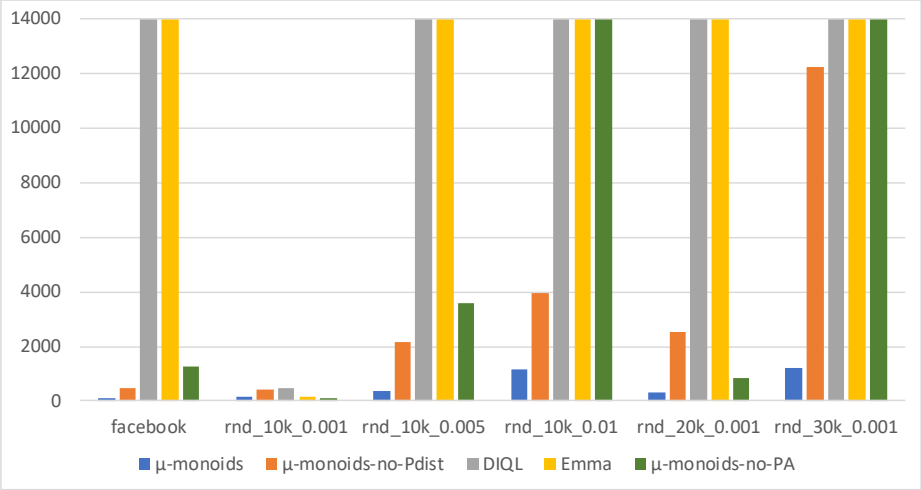


FIGURE 6. SP running times.

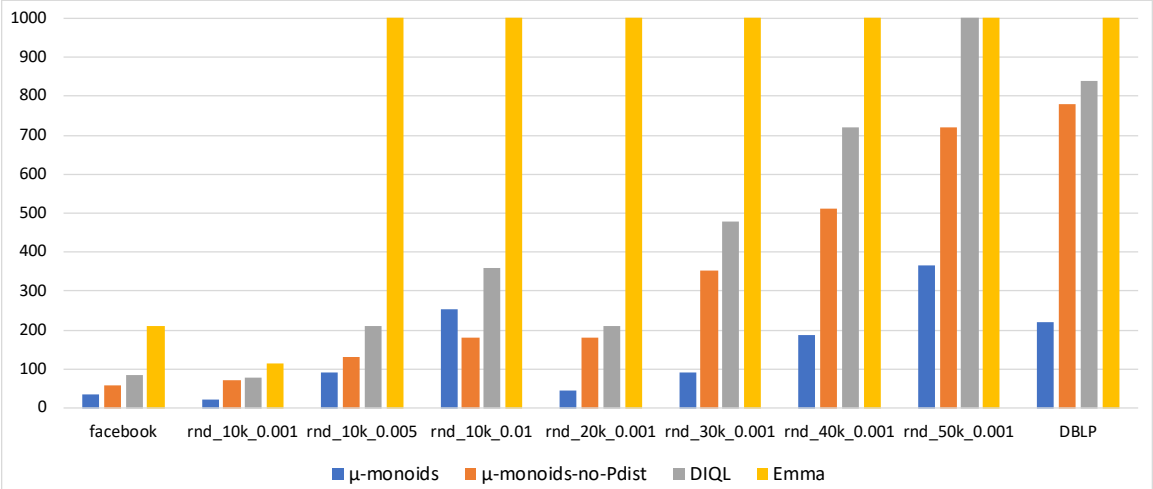


FIGURE 7. TC filter running times.

Results summary. Figures 5 to 10 show the comparison of the programs running times (reported in seconds on the y-axes). A bar reaching the maximal value on the y-axis indicates a timeout. Each data point represents the average of 5 runs.

We observe that the programs generated by μ-monoids almost systematically outperform the other program versions and always outperform the DIQL and Emma systems.

Comparison between μ-monoids and μ-monoids-no-$\mathrm{P_{dist}}$ shows the impact of the $\mathrm{P_{dist}}$ rule. It can be noticed that the speedup achieved by this rule increases with the data size. The only case where μ-monoids is slower than μ-monoids-no-$\mathrm{P_{dist}}$ is on the $rnd\_10k\_0.01$ dataset in Fig. 7. This is due to the graph topology as this dataset is the densest graph tested. This means that the size of the intermediate results generated during the fixpoint computation is large, which puts more strain on the parallel tasks. The fixed number of

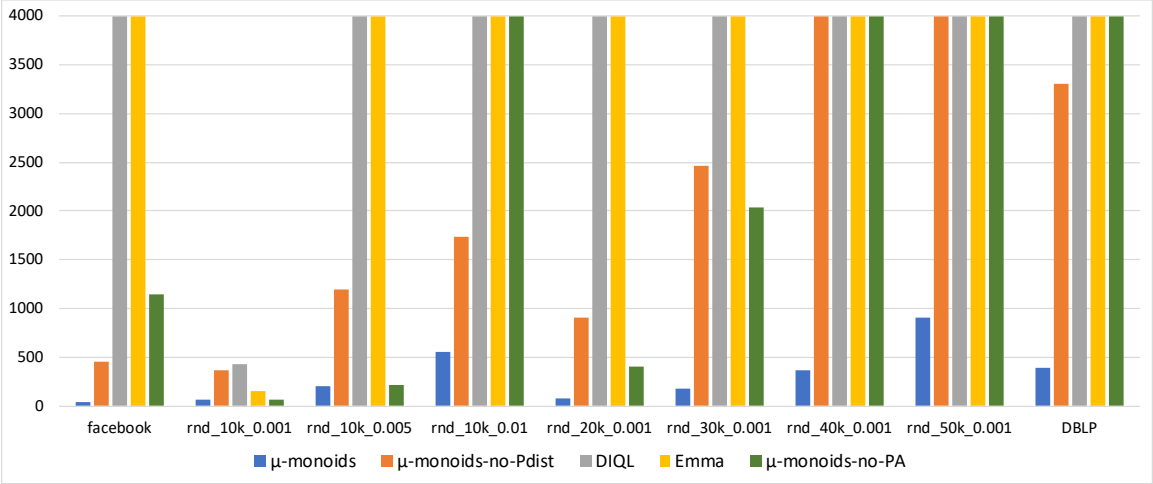


FIGURE 8. SP filter running times.

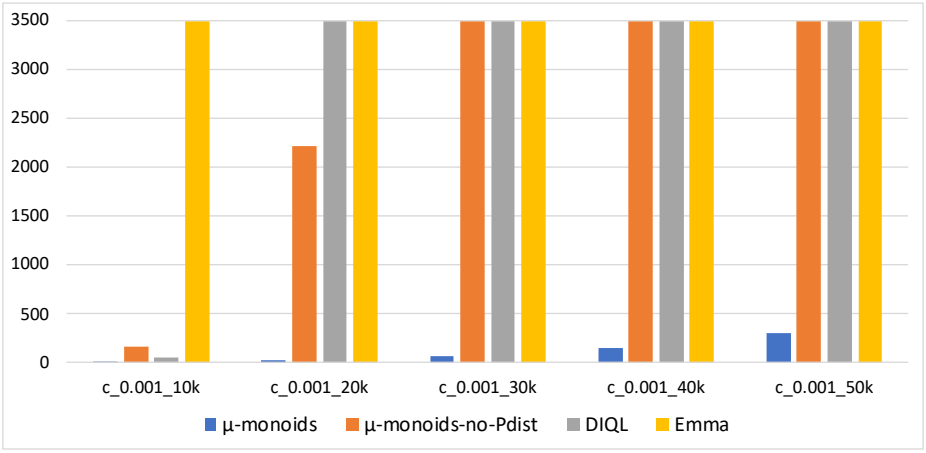


FIGURE 9. Path planning running times.

parallel tasks (Sec. 3.5) is too small. We observe in practice that, in this case, increasing the number of partitions improves the performance of the system.

In the SP, SP filter and path planning programs, both P_{dist} and PA are applied in μ-monoids. We notice that the speedup of μ-monoids in comparison to DIQL and Emma is even more important in these cases. Comparison between μ-monoids and μ-monoids-no-PA shows the impact of the PA rule alone. It can also be observed that applying PA and not P_{dist} (μ-monoids-no-P_{dist}) can be faster than applying P_{dist} and not PA (μ-monoids-no-PA) and the other way round depending on the cases. It is the combination of these two rules that leads to the best performances.

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.

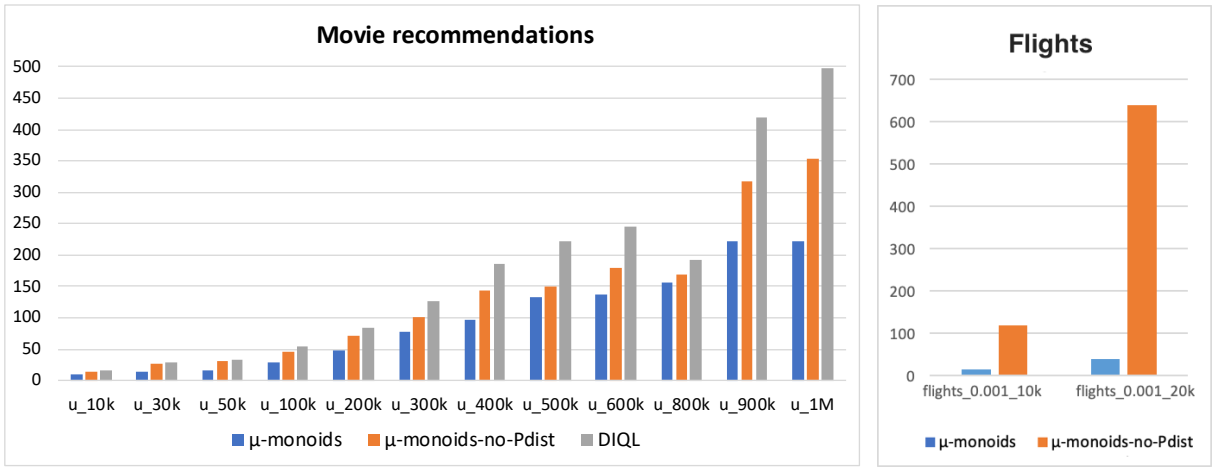


FIGURE 10. Movie recommendations and Flights running times.

5. Related works

The idea of using an intermediate representation (or an algebra) for representing user queries and performing automatic optimization originates from the work of Codd [Cod70]. He 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 algebra being widely adopted by database systems and extensively studied in database research. It also led to the standard SQL language. SQL is a *Domain Specific Language (DSL)* that is called from within a general purpose language (also called *the host language*). It gets translated to a relational algebra term that gets optimized then translated to a physical execution plan. Work in [GSM21] surveys state of the art approaches for handling iterations in distributed systems and classifies them to different categories. Among these categories are the relational algebra approach mentioned earlier, and the functional approach that offer higher-order functions for specifying control flow (such as loops). The present work belongs to this latter category. In this section we review related literature along two lines of work: works that are based on the relational model, and works that are based on more generic data models.

5.1. **Works based on the relational formalism.** The relational model is based on n-ary relations to represent entities and relationships between them. An n-ary relation is a set of rows. Each row consists of a tuple of n records of atomic types. To operate on these relations, [Cod70] proposed the relational algebra. Relational algebra offers operations on relations such as projection, selection, and join, as well as a number of rewrite rules that aim to optimize expressions regardless of their initial shape.

Another prevalent formalism based on the relational model is Datalog [MTKW18]. A Datalog program is composed of rules that infer new facts from previously known facts. Facts are expressed as predicates depending on a fixed number of variables. They can thus be seen as relations. Optimization techniques such as Magic Sets [BMSU86, SZ86] or Demand transformation [TL11] are proposed to optimize Datalog programs.

Regarding the ability to express recursion, a number of formalisms that extend RA with a recursive operator have been proposed [Agr88, AU79]. The algebra proposed in [JGGL20] provides more optimizations of recursion than previous works on RA and recursive Datalog. However, it is limited to the centralized setting and to relational algebra. Regarding distribution, the Spark SQL [AXL+15] library enables the user to write SQL queries and process relational data using `dataset`s or `dataframe`s. Queries are optimized using the Spark Catalyst engine and executed in a distributed way on the Spark platform. BigDatalog [SYI+16] is a system that studies the distribution of Datalog programs on Spark. Compared to previous distributed Datalog systems such as Socialite [SPSL13] and Myria [WBH15], it achieves better performances. The BigDatalog system uses the Datalog GPS technique [SL91] 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 other formalisms. In contrast, the present work proposes a new distribution method designed for a more generic algebra.

5.2. **Works based on more generic formalisms.** 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. In Sec. 2.1.1 we discussed the 3 types of data structures which we call *collections*: lists, bags, and sets. They are, along with other data structures, part of what is called the Boom hierarchy of types [Bun93]. The author of this paper presents data structures of that hierarchy as free algebras. A data structure value can either be empty $[\,]$, a singleton $[a]$, or a combination of two values using a binary operator $c1 +\!\!+ c2$. $+\!\!+$ can obey to a combination of four algebraic laws: unit (it has a unit element), associativity, commutativity, and idempotence. Different combinations of laws lead to different types of structures. [Bun93] defines 16 types of data structures for all possible combinations of these laws. For instance, *tree* is a data structure where only the first law is satisfied: it is not associative (and therefore not a monoid). Lists are the data structure from the hierarchy obeying only unit and associativity. Bags are obtained when we add commutativity, and sets when we add idempotence. These three structures, which are monoids, are called *collection monoids* in the work of [Feg17]. It is on this basic notion this work 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 approach [AKM19]. Both of these approaches propose an algebra for distributed collections. We review these works in more detail below.

It was shown in [BNTW95] that monads can be used to generalize nested relational algebra to different collection types and complex data. Wadler also explored and developed monad comprehensions [Wad92] and ringad comprehensions [Gib16], inspired by the early works on list comprehensions [Tur16, PJ87]. These ideas were used in LINQ [MBB06], Ferry [GMRS09], and Emma [AKKM16] which are comprehensions-based programming languages. To be evaluated, LINQ and Ferry queries get translated into an intermediate form that can be executed on relational database systems supporting SQL. As they target relational database systems, the set of host language expressions that can be used in query clauses like selection and projection is restricted. In addition, they do not analyse comprehensions to make optimizations. Emma [AKKM16] is a comprehensions-based language similar in spirit, but which rather targets JVM-based parallel dataflow engines (such as Spark and Flink). DryadLinq [YIF+08] proposes a system that distributes LINQ queries on the Dryad platform [IBY+07]. In these works recursion is expressed using host language loops that are not modelled in the algebra, hence with no optimisation provided. The idea of using monoids

and monoid homomorphisms for modeling computations with data collections originates from the works found in [TBN91, TBO91]. Fegaras proposed a monoid comprehension calculus [FM00] which later evolved in the monoid algebra presented in [Feg17]. It proposes an algebra based on monoid homomorphisms therefore with parallelism at its core: a homomorphic operation H on a collection is defined as the application of H on each subpart of the collection, results are then gathered using an associative operator. Distributed collections are modelled using the union representation of bags, and collection elements can be of any type defined in the host language. The algebra allows for defining second order operators such as flatmap and reduce that can take a UDF written in the host language as an argument. The present work further builds on this approach and proposes a generic criteria (Sec. 3.1.1) using the host language type checking system that examines those UDFs in order to determine whether the PF optimization can take place. The monoid algebra [Feg17] has a `repeat` operator, however no optimization is provided for this operator. The authors of [FN18] designed DIQL (a DSL that translates to the monoid algebra). Using reflection of the host language (Scala in this work) and quotations, queries of this DSL can be compiled and type checked seamlessly with the rest of the host language code. In fact, Emma uses the same approach as well. This approach offers more optimization opportunities than approaches like the Spark and Flink API. As argued in [AKM19], even though these APIs offer a DSL that is well integrated with the host language and allow for expressing general purpose computations, they suffer from the difficulty of automatically optimizing programs. This is due to the limited program context available in the intermediate representation of the DSLs. 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.

6. 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 introduced "μ" fixpoint operator is that 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.

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.

References

[Agr88] R. Agrawal. Alpha: an extension of relational algebra to express a class of recursive queries. *IEEE Transactions on Software Engineering*, 14(7):879–885, July 1988. doi:10.1109/32.42731.

[AKKM16] Alexander Alexandrov, Asterios Katsifodimos, Georgi Krastev, and Volker Markl. Implicit parallelism through deep language embedding. *SIGMOD Record*, 45(1):51–58, 2016. doi:10.1145/2949741.2949754.

[AKM19] Alexander Alexandrov, Georgi Krastev, and Volker Markl. Representations and optimizations for embedded parallel dataflow languages. *ACM Trans. Database Syst.*, 44(1):4:1–4:44, January 2019. URL: http://doi.acm.org/10.1145/3281629, doi:10.1145/3281629.

[AU79] Alfred V. Aho and Jeffrey D. Ullman. Universality of data retrieval languages. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 110–119, New York, NY, USA, 1979. ACM. URL: http://doi.acm.org/10.1145/567752.567763, doi:10.1145/567752.567763.

[AXL+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 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, 2015. Association for Computing Machinery. doi:10.1145/2723372.2742797.

[BMSU86] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM. URL: http://doi.acm.org/10.1145/6012.15399, doi:10.1145/6012.15399.

[BNTW95] Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995. doi:10.1016/0304-3975(95)00024-Q.

[Bun93] Alexander Bunkenburg. The boom hierarchy. In John T. O'Donnell and Kevin Hammond, editors, *Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, UK, July 5-7, 1993*, Workshops in Computing, pages 1–8. Springer, 1993. doi:10.1007/978-1-4471-3236-3_1.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015. URL: http://sites.computer.org/debull/A15dec/p28.pdf.

[Cod70] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[CWL14] Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax., february 2014. URL: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225.

[DG04] Jeffrey Dean and Sanjay Ghemawat. 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, 2004. URL: http://www.usenix.org/events/osdi04/tech/dean.html.

[diq] Diql repository. URL: https://github.com/fegaras/DIQL.

[Feg17] Leonidas Fegaras. An algebra for distributed big data analytics. *Journal of Functional Programming*, 27:e27, 2017. doi:10.1017/S0956796817000193.

[fIU19] Max Planck Institute for Informatics and Telecom ParisTech University. YAGO: A high-quality knowledge base, july 2019. URL: https://www.mpi-inf.mpg.de/yago-naga/yago/.

[FM00] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25(4):457–516, 2000. URL: http://portal.acm.org/citation.cfm?id=377674.377676.

[FN18] Leonidas Fegaras and Md Hasanuzzaman Noor. 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, 2018. doi:10.1109/BigDataCongress.2018.00008.

[Gib16] Jeremy Gibbons. Comprehending ringads - for phil wadler, on the occasion of his 60th birthday. In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on*

the Occasion of His 60th Birthday, volume 9600 of *Lecture Notes in Computer Science*, pages 132–151. Springer, 2016. doi:10.1007/978-3-319-30936-1_7.

[GMRS09] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: Database-supported program execution. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, page 1063–1066, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1559845.1559982.

[GSM21] Gábor E. Gévay, Juan Soto, and Volker Markl. Handling iterations in distributed dataflow systems. *ACM Comput. Surv.*, 54(9), oct 2021. doi:10.1145/3477602.

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 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, 2007. Association for Computing Machinery. doi:10.1145/1272996.1273005.

[JGGL20] Louis Jachiet, Pierre Genevès, Nils Gesbert, and Nabil Layaïda. On the optimization of recursive relational queries: Application to graph queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 681–697. ACM, 2020. https://hal.inria.fr/hal-01673025/document. URL: https://hal.inria.fr/hal-01673025/document, doi:10.1145/3318464.3380567.

[Les19] Jure Leskovec. Snap: Stanford large network dataset collection, november 2019. URL: https://snap.stanford.edu/data/.

[LGL20] Muideen Lawal, Pierre Genevès, and Nabil Layaïda. A cost estimation technique for recursive relational algebra. In Mathieu d'Aquin, Stefan Dietze, Claudia Hauff, Edward Curry, and Philippe Cudré-Mauroux, editors, *CIKM '20: The 29th ACM International Conference on Information and Knowledge Management, Virtual Event, Ireland, October 19-23, 2020*, pages 3297–3300. ACM, 2020. doi:10.1145/3340531.3417460.

[LV12] Leonid Libkin and Domagoj Vrgoč. 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, 2012. Association for Computing Machinery. doi:10.1145/2274576.2274585.

[Mar19] Volker Markl. Emma is a quotation-based scala dsl for scalable data analysis., november 2019. URL: https://github.com/emmalanguage.

[MB11] Erik Meijer and Gavin Bierman. A co-relational model of data for large shared data banks. *Commun. ACM*, 54(4):49–58, April 2011. URL: http://doi.acm.org/10.1145/1924421.1924436, doi:10.1145/1924421.1924436.

[MBB06] Erik Meijer, Brian Beckman, and Gavin M. Bierman. 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, 2006. doi:10.1145/1142473.1142552.

[MTKW18] David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren. *Datalog: Concepts, History, and Outlook*, page 3–100. Association for Computing Machinery and Morgan and Claypool, 2018. URL: https://doi.org/10.1145/3191315.3191317.

[OMM+04] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical report, 2004.

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[RRV17] Juan L. Reutter, Miguel Romero, and Moshe Y. Vardi. Regular queries on graph databases. *Theor. Comp. Sys.*, 61(1):31–83, jul 2017. doi:10.1007/s00224-016-9676-2.

[SL91] Jürgen Seib and Georg Lausen. 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, 1991. Association for Computing Machinery. doi:10.1145/113413.113435.

[SPSL13] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proc. VLDB Endow.*, 6(14):1906–1917, September 2013. doi:10.14778/2556549.2556572.

[SYI+16] Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big data analytics with datalog queries on spark. In Fatma Özcan, Georgia Koutrika,

and Sam Madden, 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, 2016. doi:10.1145/2882903.2915229.

[SZ86] Domenico Saccà and Carlo Zaniolo. On the implementation of a simple class of logic queries for databases. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 16–23, New York, NY, USA, 1986. ACM. URL: http://doi.acm.org/10.1145/6012.6013, doi:10.1145/6012.6013.

[TBN91] Val Tannen, Peter Buneman, and Shamim A. Naqvi. Structural recursion as a query language. In *Database Programming Languages: Bulk Types and Persistent Data. 3rd International Workshop, August 27-30, 1991, Nafplion, Greece, Proceedings*, pages 9–19, 1991.

[TBO91] Val Tannen, Peter Buneman, and Atsushi Ohori. Data structures and data types for object-oriented databases. *IEEE Data Eng. Bull.*, 14(2):23–27, 1991. URL: http://sites.computer.org/debull/91JUN-CD.pdf.

[TL11] K Tuncay Tekle and Yanhong A Liu. 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, 2011.

[Tur16] D. A. Turner. *Recursion Equations as a Programming Language*, pages 459–478. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-30936-1_24.

[Wad89] Philip Wadler. 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, 1989. Association for Computing Machinery. doi:10.1145/99370.99404.

[Wad92] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992. doi:10.1017/S0960129500001560.

[WBH15] Jingjing Wang, Magdalena Balazinska, and Daniel Halperin. Asynchronous and fault-tolerant recursive datalog evaluation in shared-nothing engines. *Proc. VLDB Endow.*, 8(12):1542–1553, August 2015. doi:10.14778/2824032.2824052.

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 1–14, USA, 2008. USENIX Association.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: a unified engine for big data processing. *Commun. ACM*, 59(11):56–65, 2016. URL: http://doi.acm.org/10.1145/2934664, doi:10.1145/2934664.

[ZYD+17] Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. Fixpoint semantics and optimization of recursive datalog programs with aggregates. *Theory Pract. Log. Program.*, 17(5-6):1048–1065, 2017. doi:10.1017/S1471068417000436.