An Algebra with a Fixpoint Operator for Distributed Data Collections

Sarah Chlyah, Nils Gesbert, Pierre Genevès, Nabil Layaïda

An Algebra with a Fixpoint Operator for Distributed Data Collections

SARAH CHLYAH, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

NILS GESBERT, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

PIERRE GENEVÈS, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

NABIL LAYAÏDA, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

We present an algebra with a fixpoint operator which is suitable for modeling computations with distributed collections found in big data frameworks. We show that under reasonable conditions this fixpoint can be evaluated by parallel loops with one final merge rather than by a global loop requiring network overhead after each iteration. We also show when and how filters can be pushed through recursive terms, proposing optimisation rules. This makes it possible to develop algebraic optimizations in the presence of recursion, in a way which is especially suited for query optimizers and compilers targeting big data frameworks.

1 INTRODUCTION

Ideas from functional programming play a major role in the construction of big data analytics applications. For instance they directly inspired Google's Map/Reduce. Big data frameworks (such as Spark [Zaharia et al. 2016] and Flink [Carbone et al. 2015]) further improved these ideas and became prevalent platforms for the development of large-scale data intensive applications. The core idea of these frameworks is to provide intuitive functional programming primitives for processing immutable distributed collections of data. Writing efficient applications is nevertheless not trivial since some primitives also hide costs related to data distribution. A typical modern analytics application combines workloads of various nature: data extraction, preparation and transformation (often performed with relational queries), iterative computations (such as recursive queries on graphs) and analytical workloads (such as statistical computations and machine learning). This makes it even harder to write globally efficient applications.

The optimisation of distributed applications has attracted a lot of attention [Armbrust et al. 2015; Fegaras and Noor 2018; Meng et al. 2016; Palkar et al. 2018; Shkapsky et al. 2016; Zadeh et al. 2016]. However, most results so far concern a specific aspect in isolation. For example, declarative relational queries are optimized in [Armbrust et al. 2015], iterative queries via datalog in [Shkapsky

Authors' addresses: Sarah Chlyah, Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, 655 avenue de l'europe, Montbonnot, 38330, France, sarah.chlyah@inria.fr; Nils Gesbert, Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, 655 avenue de l'europe, Montbonnot, 38330, France, nils.gesbert@inria.fr; Pierre Genevès, Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, 655 avenue de l'europe, Montbonnot, 38330, France, pierre.geneves@cnrs.fr; Nabil Layaïda, Tyrex team, Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, 655 avenue de l'europe, Montbonnot, 38330, France, nabil.layaida@inria.fr.

et al. 2016]; distributed variants of machine learning algorithms are implemented in [Meng et al. 2016] and optimized in [Zadeh et al. 2016]; and data traversals accross libraries are optimized in [Palkar et al. 2018]. One major challenge consists in automatically optimising a distributed application as a whole including its various workloads. Despite it is already acknowledged that some multi-aspect optimizations can improve performances by orders of magnitude (as noticed in [Palkar et al. 2018]), few attempts were made so far at optimizing holistically, involving joint optimisations of diverse workloads. One reason for that is that we lack appropriate foundations for analysing programs that process distributed data collections.

The goal of our work is to design an algebra supporting an intermediate representation to compile and optimize big data applications, that can be written using domain specific languages, relational and graph query languages, or Spark-style programs, so they can run in an optimised way on big data platforms.

Relational Algebra (RA) [Codd 1970] is a mature theory for optimizing queries very appropriate for basic data access operations. It requires extensions to deal with iterative workloads [Fegaras and Noor 2018; Shkapsky et al. 2016], analytical workloads, and distributed data. However, extensions of RA inherit two main particularities. First, in order to benefit from RA optimizations, one needs to conform to the underlying formal model (relational tables) which imposes to split data into columns. This results in the necessity to perform expensive data joins for (re)building data representations that are meaningful in applications. More often than ever, native data already come in an composite form, and there is no reason to split it into columns forcing to rejoin them afterwards, rather than simply preserving the initial native form. A second particularity of RA is that it completely abstracts over the distribution of data. Expressible logical optimisations are then naturally valid in both centralized and distributed settings. However, this limits the development of optimisations specific to the distributed setting, where performance heavily depends on data transfers and especially on the size of intermediate results to be transferred.

For designing distributed optimisation rules, we adopt an approach in which data distribution stands at the core of the algebraic formalism. This makes it possible to design optimization rules that seek to minimize data transfers by ensuring that computations remain as close as possible to where data reside. We develop our approach by further building on the algebra proposed in [Fegaras 2017; Fegaras and Noor 2018] to propose an extended algebra to optimize iterative workloads.

Contributions. Our contributions are twofold:

(1) the extension of the monoid algebra with a 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.

(2) Rule-based optimisation with new rules for optimizing in the presence of the fixpoint operator, in particular for pushing filters through a recursive terms, and for filtering inside a fixpoint before a join.

Outline. We introduce the μ-monoids algebra in Section 2, presenting its data model, syntax, semantics and evaluation rules. We present algebraic term rewriting rules in Section 3, detailing when and how they apply. We then discuss related works in Section 4 before concluding in Section 5.

2 THE μ-MONOIDS ALGEBRA

One key aspect for processing big data is to split data into partitions and distribute both data partitions and computations to several machines, in such a way that partitions are processed in parallel, intermediate results coming from different machines are combined, and a unique final result is obtained, regardless of how data was split initially. This aspect imposes a few constraints on the computations that combine intermediate results. Typically, functions used as aggregators must be associative. For that reason, algebras based on monoids are appropriate to model these distributed data-centric computations. Such an algebra provides operations that are monoid homomorphisms, which means they can be broken down to the application of an associative operator. This associativity implies that parts of the computation can actually be done in parallel and combined to get the final result.

In this section, we describe a core calculus, which we call μ-monoids, intended to model a host language (e. g. Scala) subset that is used for computations on a big data framework (through an API provided by the framework). 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 our algebraic operations. In μ-monoids, we formalize *some* of those functional constructs. The reason behind this is to characterize the set of constructs that we are able to (and that we need to) analyse in the algebraic expressions. This characterization is needed for example in the optimisation rules because they analyse the pattern and body of flatmap expressions in order to check whether the optimisation can take place.

Making explicit only the shapes that are interesting for our analysis allows us to abstract from the host language. This way, constructs of the host language other than those which we model explicitly are represented as *constants* c, as they are going to be left to the host language 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 $tf_1 \to tf_2$, it can be applied to any argument of type tf_1 to yield results of type tf_2.

We first describe the data model we consider, then in Sec. 2.2 we introduce the syntax of our core calculus and, in Sec. 2.3, a minimal type system for it. We then proceed to give a denotational semantics for our specific constructs in Sec. 2.4 and discuss evaluation of expressions in Sec. 2.5.

2.1 Data model: distributed collections of data

In big data frameworks, a data collection is divided into sub-collections stored into each machine. A collection can be in the form of a bag (a structure where the order is not important and where elements can be repeated), a list (a sequenced bag), or a set (a bag where elements do not repeat). Bags are the most suitable type for handling distributed collections. Manipulating lists would require having an order between machines, but it is achievable since big data platforms support ordering operations which have lists as output. Whereas sets require making sure that elements do not repeat across machines, which is very impractical to implement. However, a reasonable way to achieve a similar behavior is to work with bags and to use a distinct operation when it is needed to make them behave like sets. In the context of this paper and for the sake of simplicity, we will focus on bags (although the operations we present can easily be defined for lists).

The syntax of data values that we consider is defined as follows:

v	::=		value
		$\texttt{true} \mid \texttt{false}$	booleans
	$\mid$	c	constant
	$\mid$	$v_1, v_2, ..., v_n$	tuple
	$\mid$	$\{v\}$	singleton
	$\mid$	$\{v_1\} \uplus \{v_2\} \uplus ... \uplus \{v_n\}$	bag

in which a bag is seen as the union of its singletons where $\uplus$ denotes the bag union operator.

We define the following syntax for the types of values:

t_l	::=		local type
		$\mathbb{B}$	basic type
	$\mid$	$t_l \times t_l \times ... \times t_l$	product type
	$\mid$	$\texttt{Bag}_l[t_l]$	local bag type
t_d	::=		distributed type
		$\texttt{Bag}_d[t_l]$	distributed bag type
t	::=		type
		t_l	local type
	$\mid$	t_d	distributed type
	$\mid$	$t \to t$	function type

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

For a given type t, we denote by $\texttt{Bag}_l[t]$ the type of a local bag and by $\texttt{Bag}_d[t]$ 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 to have nested collections. We allow data distribution only at the top level though (distributed bags cannot be nested).

Some operators over bags can be defined similarly, regardless of whether bags are local or distributed. For convenience, we thus denote the type of general bags, either distributed or not as:

$$\texttt{Bag}[t] \quad ::= \quad \texttt{Bag}_l[t] \mid \texttt{Bag}_d[t]$$

For example, the bag union operator $\uplus : \texttt{Bag}[t] \times \texttt{Bag}[t] \to \texttt{Bag}[t]$ is defined for both local and distributed bags. We consider that whenever any argument of $\uplus$ is a distributed bag then the result is a distributed bag as well; if, on the contrary, both arguments are local bags then the result may be either a local bag or a distributed bag. We usually do not need to distinguish the cases, but when it is relevant to do so (as in Sec. 2.5.2), we use a vertical bar | to indicate nonlocal union of local bags into a distributed one.

2.2 μ-monoids syntax

We consider patterns:

pat	::=		pattern
		$\texttt{var}$	pattern variable
	$\mid$	$(pat_1, pat_2, ..., pat_n)$	

The syntax of expressions is defined in Figure 1.

e	::=		expression
		v	value
	|	$\texttt{var}$	variable
	|	$\lambda\ pat.e$	closure
	|	$e\ e$	application
	|	$\{e\}$	singleton
	|	$e_1, e_2, ..., e_n$	tuple constr.
	|	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	conditional
	|	$\texttt{flatmap}(e, e)$	
	|	$\texttt{cogroup}(e, e)$	
	|	$\texttt{groupby}(e)$	
	|	$\texttt{reduce}(e, e)$	
	|	$\mu\ (e, e)$	fixpoint

Fig. 1. Syntax for terms.

We consider the following syntactic sugar:

$$\texttt{join}(e_x, e_y) \stackrel{\text{def}}{=} \texttt{flatmap}(\lambda(k, (s_x, s_y)).\ \texttt{flatmap}(\lambda x.\ \texttt{flatmap}(\lambda y.\ \{(k, (x, y))\}, s_y), s_x), \texttt{cogroup}(e_x, e_y))$$

μ-monoids expressions consist of functional expressions and algebraic operations (flatmap, groupby, ...) performed on collections.

Functional expressions can appear as arguments of those algebraic operations. For example $\texttt{flatmap}(\lambda a.\ \{a + 1\}, b)$ has two arguments, a functional argument $\lambda\ a.\{a + 1\}$ which represents a lambda expression (a function that returns a singleton of the incremented value of its argument), and a second argument b which is a variable expression that references some collection. Aside from lambda expressions and variables, a functional expression can also be a function application, a singleton, a tuple or a value.

2.3 Well-typed terms

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

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

We first define the environment obtained by matching a data type to a pattern by the following:

$$\frac{}{match(\texttt{var}, tf) \to \texttt{var} : tf} \qquad \frac{\forall i\ match(pat_i, t_i) \to \Gamma_i}{match((pat_1, \ldots, pat_n), t_1 \times \cdots \times t_n) \to \Gamma_1 \cup \cdots \cup \Gamma_n}$$

$$\frac{\Gamma \vdash e_1 : t \to T_1[t'] \qquad \Gamma \vdash e_2 : T_2[t] \qquad T_1 = T_2 \vee (T_1 = \mathsf{Bag}_l \wedge T_2 = \mathsf{Bag}_d)}{\Gamma \vdash \mathtt{flatmap}(e_1, e_2) : T_2[t']}$$

$$\frac{\Gamma \vdash e_1 : T[t \times t_1] \qquad \Gamma \vdash e_2 : T[t \times t_2]}{\Gamma \vdash \mathtt{cogroup}(e_1, e_2) : T[t \times (\mathsf{Bag}_l[t_1] \times \mathsf{Bag}_l[t_2])]}$$

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

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

$$\frac{\Gamma \vdash e_1 : T[t] \qquad \Gamma \vdash e_2 : T[t] \to T[t]}{\Gamma \vdash \mu\,(e_1, e_2) : T[t]} \qquad \frac{\forall i\ \Gamma \vdash e_i : t_i}{\Gamma \vdash e_1, e_2, \ldots, e_n : t_1 \times t_2 \times \cdots \times t_n}$$

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash \{e\} : \mathsf{Bag}_l[t]} \qquad \frac{match(pat, t') \to \Gamma' \qquad \Gamma + \Gamma' \vdash e : t}{\Gamma \vdash \lambda\ pat.e : t' \to t} \qquad \frac{\Gamma \vdash e_1 : t \to t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'}$$

$$\frac{\Gamma(\mathsf{var}) = tf}{\Gamma \vdash \mathsf{var} : tf} \qquad \frac{type(c) = tf}{\Gamma \vdash c : tf} \qquad \frac{\Gamma \vdash e : bool \qquad \Gamma \vdash e_1 : tf \qquad \Gamma \vdash e_2 : tf}{\Gamma \vdash \mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : tf}$$

Fig. 2. Typing judgements.

If, according to these rules, there is no Γ such that $match(pat, tf) \to \Gamma$ holds, we say that pattern pat is incompatible with type tf. 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.

A term e is well-typed in a given environment Γ iff $\Gamma \vdash e : \tau$ for some type $\tau \in \{t, t_d\}$, as judged by the relation defined in Figure 2. 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 host language. An interesting particular case, however, is when a λ expression is used directly as the first parameter of a `flatmap`(,) or `reduce`(,). We can write the following compound deterministic rule in the case of `flatmap`(,), for example:

$$\frac{\Gamma \vdash e_2 : T_2[t] \qquad match(pat, t) \to \Gamma' \qquad \Gamma + \Gamma' \vdash e_1 : T_1[t'] \qquad T_1 = T_2 \vee (T_1 = \mathsf{Bag}_l \wedge T_2 = \mathsf{Bag}_d)}{\Gamma \vdash \mathtt{flatmap}(\lambda\ pat.e_1, e_2) : T_2[t']}$$

2.4 μ-monoids denotational semantics

Monoid homomorphisms. The semantics of the μ-monoids algebra extends the semantics of the monoid algebra [Fegaras 2017]. We first recall basic definitions for the monoid algebra and then present the fixpoint operation that we introduce.

Briefly, a *monoid* is an algebraic structure $(S, \oplus, e)$ where S is a set (called the carrier set of the monoid), $\oplus$ an associative binary operator between elements of S, and e is an identity element for $\oplus$. A monoid homomorphism h from $(S, \oplus, e)$ to $(S', \otimes, e')$ is a function $h : S \to S'$ such that $h(x \oplus y) = h(x) \otimes h(y)$ and $h(e) = e'$.

Given any type α, we can consider the set of lists (finite sequences) of elements of α; if we equip this set with the concatenation operator, it yields a monoid (algebraically called the free monoid on α), whose identity element is the empty list. Let $U_{\texttt{List}} : \alpha \to \texttt{List}[\alpha]$ be the singleton construction function. This monoid has the following universal property: let $(S, \otimes, e)$ be any monoid and $f : \alpha \to S$ any function, then there exists exactly one monoid homomorphism $H_f^{\otimes} : \texttt{List}[\alpha] \to S$ such that $H_f^{\otimes} \circ U_{\texttt{List}} = f$. This homomorphism can be simply defined by $H_f^{\otimes}([a_1, \ldots, a_n]) = f(a_1) \otimes \cdots \otimes f(a_n)$.

Then, consider a set of algebraic laws for a binary operator, for example commutativity ($a \otimes b = b \otimes a$) or the 'graphic identity' $a \otimes b \otimes a = a \otimes b$. It is possible to define the quotient of the list monoid by a set of such laws. For example, the congruence induced by commutativity relates all lists which contain exactly the same elements in different orders; i. e. the quotient of the monoid $\texttt{List}[\alpha]$ by commutativity is isomorphic to $(\texttt{Bag}[\alpha], \uplus, \{\})$, the bag monoid, where $\uplus$ is the bag union operator and $\{\}$ the empty bag. Such quotients of the free monoid have been termed *collection monoids* by Fegaras et al. Another notable example of collection monoid is the monoid of finite sets on α, $(\texttt{Set}[\alpha], \cup, \emptyset)$, obtained by quotienting with both commutativity and the graphic identity[1].

Collection monoids inherit the universal property of lists in the following way: let $(T[\alpha], \oplus, e_T)$ be a collection monoid noted $(\oplus)$ and $(\beta, \otimes, e)$ a monoid noted $(\otimes)$, where α and β are arbitrary types. *Suppose $\otimes$ obeys all the algebraic laws of $\oplus$,* then for any function $f : \alpha \to \beta$, there exists a unique homomorphism[2] $H_f^{\otimes} : T[\alpha] \to \beta$ such that $H_f^{\otimes} \circ U_T = f$.

As a slight abuse of notation, we will sometimes refer to 'the monoid $\uplus$' for example to designate the monoid of bags on an unspecified type α.

This formalism allows expressing the distribution and the parallelisation of computations. The reason is that it suggests that the result of a computation (application of a monoid homomorphism) on a collection can be obtained by dividing the collection into subcollections (eventually singletons), applying the computation on each subcollection, and combining the results with an associative and commutative (in the case of bags) operator. The associativity of the operator makes it possible for each worker node to combine its sub-results locally without leading to erroneous results, and commutativity means that the nodes need not be ordered[3].

As mentioned previously in 2.1, we will focus on bag monoids (local and distributed bags) as a start monoid for the homomorphic operations. Those operations share the same denotational semantics for local and distributed bags (they return the same values regardless of whether those values are distributed or not).

[1] As a less notable example, the graphic identity alone yields the monoid $\texttt{OrderedSet}[\alpha]$.
[2] It may be worth mentioning that, thanks to this property, while a collection monoid $T[\alpha]$ has the structure of a monoid, the constructor T itself also defines a *monad*, whose unit function is U_T (singleton construction) and whose monadic operations *map* and *flatmap* can be defined from the universal property.
[3] As mentioned earlier, it would be possible to work on lists, and then the operators would not need to be commutative.

To summarise in the case of bags, if $(\beta, \otimes, e)$ is a commutative monoid and $f : \alpha \to \beta$ is any function, the monoid homomorphism $H_f^{\otimes}$ from $\uplus$ to $\otimes$ satisfies the following:

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

Figure 3 gives the denotational semantics of the main algebraic operations.

flatmap operator. We consider a function $f : \alpha \to \mathtt{Bag}[\beta]$. $\mathtt{flatmap}(f, X)$ applies f on each element in the bag X and returns a dataset that is the union of all results.

This operation is a monoid homorphism $H_f^{\uplus}$ from $(\mathtt{Bag}[\alpha], \uplus, \{\})$ to $(\mathtt{Bag}[\beta], \uplus, \{\})$.

$$\mathtt{flatmap}(f, A) = \biguplus_{a \in A} f(a)$$
$$\mathtt{groupby}(A) = \{(k, \{v \mid (k, v) \in A\}) \mid k \in keys(A)\}$$
$$\mathtt{cogroup}(A, B) = \{(k, (\{v \mid (k, v) \in A\}, \{w \mid (k, w) \in B\})) \mid k \in keys(A) \cup keys(B)\}$$
$$\mathtt{reduce}(f, A) = r_N, \text{ where } A = \{a_1, a_2, ..., a_N\}, r_n = f(r_{n-1}, a_n), r_1 = a_1$$
$$\mu(R, \varphi) = \bigcup_{n \in N} \varphi^{(n)}(R), \text{ where } f : X \to X \cup \varphi(X)$$

Where:

– The comprehensions used denote bag comprehensions
– $keys(A) = \text{distinct}(\{k \mid (k, a) \in A\})$
– $\cup$ is distinct union of bags

Fig. 3. Denotational semantics

reduce operator. $\mathtt{reduce}(\oplus, X)$ reduces the elements of the input dataset by combining them with the $\oplus$ operator, which must be associative and commutative.

This operation is a monoid homomorphism $H_{id}^{\oplus}$ from the monoid $\uplus$ to the monoid $\oplus$.

groupBy operator. groupBy takes a bag of elements in the form (k, v) and returns a bag of elements in the form (k, V) where V is the bag of all elements having the same key k in the input dataset. For example, $\mathtt{groupby}(\{(1, 2), (1, 4), (2, 2), (2, 1), (1, 3)\}) = \{(1, \{2, 4, 3\}), (2, \{2, 1\})\}$

groupBy is a monoid homorphism $H_f^{\uparrow}$ from $(\mathtt{Bag}[(\alpha, \beta)], \cup, \{\})$ to $(\mathtt{Bag}[(\alpha, \mathtt{Bag}[\beta])], \uparrow, \{\})$, where:

$$f : (\alpha, \beta) \to \mathtt{Bag}[(\alpha, \mathtt{Bag}[\beta])]$$
$$(k, v) \mapsto \{k, \{v\}\}$$

and the operator $\uparrow$ is defined such that $\{(k, b_1\}) \uparrow \{(k', b_2)\} = \begin{cases} \{(k, b_1 \uplus b_2)\} & \text{if } k = k' \\ \{(k, b_1), (k', b_2)\} & \text{otherwise} \end{cases}$

Note: Because of the way the function f is defined, the groupby operation supports bags of arbitrary types.

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

cogroup is a binary homomorphism $H^{\updownarrow}_{f_1,f_2}$ from the monoids $\uplus$ and $\uplus$ to the monoid $\updownarrow$, defined in the following way:

$$\texttt{cogroup}(X_1 \uplus Y_1, X_2 \uplus Y_2) = \texttt{cogroup}(X_1, Y_1) \updownarrow \texttt{cogroup}(X_2, Y_2)$$
$$\texttt{cogroup}(\{x\}, \{y\}) = f_1(x) \updownarrow f_2(y)$$

where:

$$f_1 : (\alpha, \beta) \to \texttt{Bag}[(\alpha, (\texttt{Bag}[\beta]), \texttt{Bag}[\gamma]))]$$
$$(k, v) \mapsto \{(k, (\{v\}, \{\}))\}$$
$$f_2 : (\alpha, \gamma) \to \texttt{Bag}[(\alpha, (\texttt{Bag}[\beta], \texttt{Bag}[\gamma]))]$$
$$(k, v) \mapsto \{(k, (\{\}, \{v\}))\}$$

and

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

Fixpoint operator. Let us first comment on the choice of our fixpoint operator. We could define mathematically a more generic operation $\mu(\psi)$ as the smallest fixpoint of the function $f : X \to X \cup \psi(X)$. $\mu(\psi)$ then consists on recursively applying f at each iteration on the result of the previous iteration starting from X = $\emptyset$ until the fix point is reached and no more new results are added.

This fixpoint operation can be defined for $\psi : \texttt{Set}[\alpha] \to \texttt{Set}[\beta]$ as well as for $\psi : \texttt{Bag}[\alpha] \to \texttt{Bag}[\beta]$, where in the first case $\cup$ is set union and in the second it corresponds to a bag union with no duplicates.

In the context of bags, it can be shown (see Appendix A.1) that when $\psi = R \uplus \varphi$ for some R, where φ is a monoid homomorphism from $\uplus$ to $\uplus$ ($\uplus \to \uplus$), the fixpoint exists (f has a fixpoint) and can be reached from the successive application of φ starting from the empty bag {}. We thus restrict our language to this particular kind of fixpoint, and we use the syntax $\mu(R, \varphi)$ to denote $\mu(R \uplus \varphi)$.

Any composition of homomorphisms $\uplus \to \uplus$ is also a homomorphism $\uplus \to \uplus$. Since we know that flatmap belongs to this kind of homomorphisms, and we can show that for join as well (in Appendix A.2), then it is sufficient for φ to be a composition of those operations as described by the following terms T (X) (X is the recursion variable):

$T(X)$	::=		
		X	
	|	$\texttt{flatmap}(f, T(X))$	f constant in X
	|	$join(T(X), A)$	A constant in X
	|	$join(A, T(X))$	A constant in X

Under this criteria, the μ operator can be seen as a monoid homomorphism $H^{\cup}_{f}$ from $\uplus$ to $\cup$, where $f(a) = \mu(\{a\}, \varphi)$:

$$\mu(R_1 \uplus R_2, \varphi) = \mu(R_1, \varphi) \cup \mu(R_2, \varphi)$$

2.5 Evaluation of expressions

2.5.1 *Local execution.*

Function application. A lambda expression $\lambda pat.\ e$ contains a pattern pat and a return expression e. When this lambda expression is applied on an argument a ($\lambda\ pat.e\ a$), the argument is matched against the pattern in the following way: if a is a tuple $(a_1, a_2, ..., a_n)$ and pat is of the form $(pat_1, pat_2, ..., pat_n)$, each a_i is recursively matched against the subpattern pat_i. If a is matched against a pattern variable p, p will take the value of a. Finally, e is evaluated by substituting the pattern variables it contains by the values affected to them during pattern matching.

Monoid homomorphisms. The definition of algebraic operations as monoid homomorphism suggests that they can be evaluated in the following way: $H_f^{\otimes}(\{v_1\} \uplus \{v_2\} \uplus ... \uplus \{v_n\}) \rightarrow f(v_1) \otimes f(v_2) \otimes ... \otimes f(v_n)$. 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. To evaluate $\mu\ (R, \varphi)$, first $\varphi(R)$ is computed. If $\varphi(R)$ is included in R, in the sense of set inclusion, then the computation terminates and the result is the distinct values of R (returning distinct values of R is especially needed if the computation terminates at the first iteration). If not, then the values from $\varphi(R)$ are added to R (using $\cup$) and we loop by evaluating $\mu\ (R \cup \varphi(R), \varphi)$. This is summarised by the following reduction rules:

$$\frac{\varphi(R) \subset R}{\mu(R, \varphi) \rightarrow \text{distinct}(R)} \qquad \frac{\varphi(R) \not\subset R}{\mu(R, \varphi) \rightarrow \mu(R \cup \varphi(R), \varphi)}$$

Note: The fixpoint operation can be computed in a more efficient way by iteratively applying φ to only the new values generated in the previous iteration. This terminates when no new values are added. Applying φ to only the new values and not to the whole set is correct thanks to the homomorphism property of φ (we have $\varphi(X \cup \varphi(X)) = \varphi(X \cup (\varphi(X) \setminus X)) = \varphi(X) \cup \varphi((\varphi(X) \setminus X))$). The algorithm is as follows:

```
res = R
new = R
while new ≠ ∅:
    new = φ(new)\res
    res = res ∪ new
return res
```

2.5.2 *Distributed execution.*

We consider in a distributed setting that distributed bags are partitionned. 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:

- $H_f^{\uplus}(R_1|R_2|...|R_p) \rightarrow H_f^{\uplus}(R_1)|H_f^{\uplus}(R_2)|...|H_f^{\uplus}(R_p)$ (partitionning does not have to change)
- $H_f^{\otimes}(R_1|R_2|...|R_p) \rightarrow H_f^{\otimes}(R_1) \otimes_{nl} H_f^{\otimes}(R_2) \otimes_{nl} ... \otimes_{nl} H_f^{\otimes}(R_p)$, 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` 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 operations like join and groupByKey.

Distribution of the fixpoint operations. As mentioned previously (Sec. 2.4), φ is a monoid homomorphism $\uplus \rightarrow \uplus$. This means that for $R = R_1 \uplus R_2$, $R \cup \varphi(R) = (R_1 \uplus R_2) \cup (\varphi(R_1) \uplus \varphi(R_2)) = R_1 \cup R_2 \cup \varphi(R_1) \cup \varphi(R_2)$.

The distributed version of the fixpoint rule would normally be:

$$\mu(R_1|R_2|..., \varphi) \rightarrow \mu((R_1 \cup \varphi(R_1)) \ \cup_{nl} \ (R_2 \cup \varphi(R_2)) \ \cup_{nl} ..., \varphi).$$

Following this rule, at each iteration, $(R_1 \cup \varphi(R_1)) \ \cup_{nl} \ (R_2 \cup \varphi(R_2)) \ \cup_{nl} ...$ will be evaluated, φ applied on the result and the stopping condition ($\varphi(R) \subset R$) checked.

Alternatively, if we use the fact that $\mu(R, \varphi)$ is a monoid homomorphism, then we can apply the following rule to evaluate it:

$$\mu(R_1|R_2|..., \varphi) \rightarrow \mu(R_1 \cup \varphi(R_1), \varphi) \ \cup_{nl} \ \mu(R_2 \cup \varphi(R_2), \varphi) \ \cup_{nl} ...$$

This execution plan will avoid doing non local set unions between all partitions at each iteration of the fixpoint. Instead, the fixpoint is executed locally (following the local rule shown previously) on each partition on a part of the input, after which set union $\cup_{nl}$ is computed once to gather results. 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.

3 OPTIMISATIONS

3.1 Pushing filter inside a fixpoint

Let us consider a term of the form:

A = `flatmap`($\lambda\pi$. `if` $c(a)$ `then` $\{\pi\}$ `else` $\{\}$, D)

The term A corresponds to the dataset D filtered based on the condition c that depends on the variable a. π is the pattern that matches each element of D and extracts the variable a from it that will be used to either keep the element or discard it.

So we can say that given any $d \in D$, $d \in A \Leftrightarrow c(\pi_a(d))$, where $\pi_a(d)$ denotes the extraction of the pattern variable a from d by the pattern π.

Let us take A = `flatmap`($\lambda\pi$. `if` $c(a)$ `then` $\{\pi\}$ `else` $\{\}$, $\mu(R, \varphi)$)

We want to explore sufficient conditions to push the filter before the fixpoint operation $\mu(R, \varphi)$. The desired goal is to find a generic way of analysing φ expressions regardless of which operations they contain (that could be generic map operations with user defined functions) and putting minimal restrictions on them that guarantee that the filter can be pushed safely.

We can show that if the following condition, noted (C) : $\forall r \in R \quad \forall s \in \varphi(\{r\}) \quad \pi_a(r) = \pi_a(s)$ is satisfied, then the filter can be pushed, and the expression A would be equivalent to $\mu(R_f, \varphi)$, where

R_f = `flatmap`$(\lambda\pi.$ `if` $c(a)$ `then` $\{\pi\}$ `else` $\{\}, R)$. In other words, the constant part R can be filtered first before applying the fixpoint on it.

This condition means that the operation φ does not change the value of a (the variable on which the filter depends), so each record of 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.

If the filter depends on multiple variables, we would need to check that the condition is fulfilled for each of them.

Proof. We show that A = $\mu(R_f, \varphi)$ when the condition (C) is fulfilled.

We note (*) the following proposition: $\forall s \in \mu(R, \varphi) \quad \exists r \in R \quad \pi_a(r) = \pi_a(s)$

(*) is verified when the condition (C) is fulfilled, and can be shown using the following: $\forall s \in \mu(R, \varphi) \quad \exists r \in R \quad \exists n \in \mathbb{N} \quad s \in \varphi^{(n)}(\{r\})$ (shown in Appendix A.1.1).

(1) $\mu(R_f, \varphi) \subset A$:

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

Let $s \in \mu(R_f, \varphi)$

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

So $c(\pi_a(s)) = c(\pi_a(r))$(= true because $r \in R_f$)

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

(2) $A \subset \mu(R_f, \varphi)$:

Let $s \in A, \quad s \in \mu(R, \varphi)$ and $c(\pi_a(s)) =$ true

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

So $c(\pi_a(r)) = c(\pi_a(s)) =$ true

So $r \in R_f$, which means distinct$(\varphi^{(n)}(\{r\})) \subset \mu(R_f, \varphi)$ (see fixpoint related proofs in Appendix A.1.1), so $s \in \mu(R_f, \varphi)$

In order to verify that condition (C) is fulfilled, one way is to use the scala type inference system on the operation φ after translating it to a scala function. By giving this function a polymorphic input type and checking the output type, we could see where the variable a is in the output and whether it was changed.

Verifying (C): $\forall r \in R \quad \forall s \in \varphi(\{r\}) \quad \pi_a(r) = \pi_a(s)$ using types. We will start first by explaining the main idea behind this before going into the details. 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. Since the type T of R elements matches the pattern π_a (otherwise the term would not be type correct, see section 2.3), we expect their tree structures to match as well (we formalize this notion below using the 'correspondsTo' relation). So we can find in T the type t corresponding to the pattern variable a by following the structure of π (against the stucture of T). If we can affirm that this data of type t is not modified (or displaced) by the φ function, then

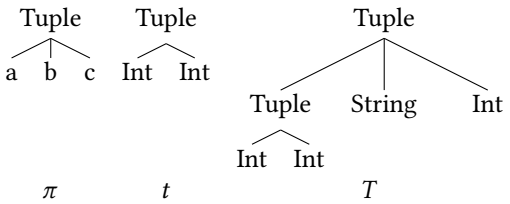
the extraction of a by the pattern π from r and from $\varphi(\{r\})$ results would lead to the same value, hence (C). For this, we use type checking and polymorphic types to follow the data of type t in the φ function. The idea behind this is the fact that if $f : t \to t$ where f is a polymorphic function and t a polymorphic type, then f has to be the identity function (f did not manipulate the value of its argument).

Definition of the 'correspondsTo' relation. Let T be a tree and n a node in this tree. We identify the path to n in T denoted path(n, T) by the sequence of pairs (number, label) that correspond to the ordered sequence of node label and next child arity for each node that is visited to reach n from the root of the tree.

Let T and T' be two trees. We define a relation in the set of T and T' nodes, denoted correspondsTo(T, T') by the following: Two nodes n_1 and n_2 are related (n_1 correspondsTo(T, T') n_2) when path(n_1, T) = path(n_2, T'). If two trees have the same structure, every node belonging to a tree will be related to a node in the other tree.

Let a be a pattern variable in π. Because π matches T, a correspondsTo(π, T) t for some node t in T. For example:

a in the pattern π corresponds to the subtree t (or more precisely the root of the subtree) in the type T, a correspondsTo(π, T) t



We consider the parametric type $C(t)$ obtained by replacing the mentionned subtree in T by the type parameter t. In the example $C(t) = (t \times String \times Int)$

We have $\forall e : C(t) \quad \pi_a(e) : t$ and e: C($\{\pi_a(e)\}$), where $\{\pi_a(e)\}$ is the singleton type containing one element $\pi_a(e)$

We then translate the φ function to Scala, give it the polymorphic input type $List[C(t)]$ (with t at the position of the variable a on which the filter depends), and use the type inference system to compute the output type. In case the function φ performs any specific operation (like addition or field access) on the element of type t other than copying it, a compilation error will be generated. Otherwise, the parameter t will appear in the output type. If the type inference system computes a return type $List[C'(t)]$, it means that for every arbitrary type t, φ takes an element of type $List[C(t)]$ and returns an element of type $List[C'(t)]$. So we have the following:

$\forall r \in R \quad r : C(\{\pi_a(r)\})$ So $\{r\} : List(C(\{\pi_a(r)\}))$ So $\varphi(\{r\}) : List(C'(\{\pi_a(r)\}))$ So $\forall s \in \varphi(\{r\}) \quad s : C'(\{\pi_a(r)\}$

So if a correspondsTo(π, $C'(t)$) t, then $\pi_a(s) : \{\pi_a(r)\}$ meaning $\pi_a(r) = \pi_a(s)$

Conclusion: In order for our filter pushing condition to be valid, it is sufficient for φ to have an output type $List(C'(t))$ when given an input of type $List(C(t))$ (with the type parameter t in $C(t)$ corresponds to a in π) and that t in $C'(t)$ corresponds to a in π. This means in other words that φ did not change the value nor the position of a in the data it manipulates, so the filter can be

performed on the data before applying φ multiple times, since this application will only generate a value with the same a for each of its input elements.

Filters depending on multiple variables. We showed that (C): $\forall r \in R \quad \forall s \in \varphi(\{r\}) \quad \pi_a(r) = \pi_a(s)$ is sufficient for pushing the filter in a fixpoint when the filter condition depends on a.

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

3.2 Filtering inside a fixpoint before a join

Let us consider the expression: join(A,B), where A is a constant and B = $\mu(R, \varphi)$. After the execution of the fixpoint, the result is going to be joined with A, so only elements of this result sharing the same keys with A are going to be kept. So in order to optimise this term, we want to push a filter that only keeps 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(B)), where F(B) = $\{(k, v) \mid (k, v) \in B, \exists w\ (k, w) \in A\}$

(2) we show that F(B) is a filter on B, that can be pushed when the criteria on pushing filters is fulfilled

(3) we show as well that

$$\forall R, F(R) = \texttt{flatmap}(\lambda(k, (s_v, s_w)).\ \texttt{if}\ s_w \neq \{\}\ \texttt{then}\ \texttt{flatmap}(\lambda v.\ \{(k, v)\}, s_v)\ \texttt{else}\ \{\}, \texttt{cogroup}(R, A))$$

3.2.1 *Inserting F before the join.*

We have $join(A, B) = \{(k, (x, y)) \mid (k, x) \in A, (k, y) \in B\}$ (proof in Appendix A.4)

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

3.2.2 *Proving that F is a filter.*

We can show that

$$\begin{aligned} F(B) &= \{(k, v) \mid (k, v) \in B, \exists w (k, w) \in A\} \\ &= \texttt{flatmap}(\lambda(k, v).\ \texttt{if}\ c(k)\ \texttt{then}\ \{(k, v)\}\ \texttt{else}\ \{\}, B) \end{aligned}$$

where $c(k)$ is the boolean expression that corresponds to the predicate $\exists w (k, w) \in A$

This expression can be: $c(k) = \texttt{reduce}(\vee, \texttt{flatmap}(\lambda(k', a).\ k == k', A))$

Which means in case φ fulfills the criteria for pushing filters we will have F(B) = F($\mu(R, \varphi)$) = $\mu(F(R), \varphi)$

3.2.3 *Rewriting the filter with a cogroup.*

We show that $F(B) = C$ where

$$C = \texttt{flatmap}(\lambda(k,(s_x,s_y)).\ \texttt{if}\ s_y \neq \{\}\ \texttt{then}\ \texttt{flatmap}(\lambda x.\ \{(k,x)\}, s_x)\ \texttt{else}\ \{\}, \texttt{cogroup}(B,A))$$

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

Let $e \in C$. So $\exists (k,(s_x,s_y)) \in \texttt{cogroup}(B,A))$ such that $\exists x \in s_x \quad e = (k,x)$ and $s_y \neq \{\}$. We have $(k,(s_x,s_y)) \in \texttt{cogroup}(B,A)$, 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 $e = (k,x) \in F(B)$.

Let $(k,x) \in F(B)$. We have $(k,x) \in B$ and $\exists w\ (k,w) \in A$. So $k \in keys(A) \cup keys(B)$. Let $s_x = \{v \mid (k,v) \in B\}$ and $s_y = \{v \mid (k,y) \in A\}$, so $(k,(s_x,s_y)) \in \texttt{cogroup}(B,A)$. Since $x \in s_x$ and $s_y \neq \{\}$ (because $\exists w\ (k,w) \in A$), then $(k,x) \in C$.

3.3 Application of rules

Given an expression e in the language, optimisation rules are going to be applied as follows:

(1) We start by pushing all the filters once

(2) We look for all subexpressions of the form $join(A, \mu(R,\varphi))$ (or $join(\mu(R,\varphi), A)$) and try to execute the optimisation described in Sec. 3.2. If φ satisfies the criteria for pushing the filter, we transform the expression to $join(A, \mu(F(R),\varphi))$

(3) On the result expression, we try to repeatedly push all filters as explained earlier until we cannot push filters any further.

(4) We rewrite F(R) coming from pushing joins to a cogroup as shown earlier.

Step 1 can be useful in case there is in e a filter on a fixpoint that is blocking the join optimisation to happen, so we try to push it first.

Because of the nature of these optimisation rules, this process is sufficient to get the best expression (according to those rules) from the initial one. The reason is that no operation can enter inside a join or a filter, so if the described process is followed, all the optimisations that can be generated from them will occur.

An interesting aspect is to characterize under which conditions those rules produce terms that are more efficient in practice.

A crucial aspect to take into account to evaluate the cost of a term is the size of non-local data transfers it generates. Another important aspect we need to consider as well is the local complexity of the term.

A filter reduces the size of intermediate data. Hence, the size of transfers made by operators working on those intermediate data is going to be smaller. The operators are also going to be executed faster since the data they manipulate is smaller. This means that pushing filters the furthest possible is only going to improve the two aforementioned metrics (or not have any impact in case the filter does not remove any element).

The rule "filtering inside a fixpoint before a join" however, incurs an additional cogroup to compute the filter that is then going to be pushed in the fixpoint (as detailed in Sec. 3.2).

We consider that, in general, this rule improves local complexity. The reason is that the additional cogroup represents a one time cost, whereas the pushed filter makes R (the first argument of the fixpoint $\mu(R, \varphi)$) smaller. Therefore each iteration of the fixpoint will execute faster as it will deal 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 is also going to execute faster because its size is reduced prior to the join. We can then consider that, in general, the additional cogroup cost is compensated by the speedup of each iteration in the fixpoint as well as the final join.

In order to analyse the impact of the rule on non-local data transfers, we need to estimate and compare the size of transfers incurred by the terms: $join(A, \mu(R, \varphi))$ and $join(A, \mu(F(R), \varphi))$ (obtained after applying the rule). As mentionned in (Sec. 2.5.2), all our algebraic operators apart from flatmap trigger non-local transfers. We then consider the following, where $\texttt{size}_t(e)$ is the size of transfers incurred by the term e:

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

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

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

$S_2 \approx o(2 \times \texttt{size}(A) + 2 \times \texttt{size}(\mu(F(R), \varphi)) + \texttt{size}(R))$, here $F(R)$ 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.

So, in order to know whether this optimisation rule improves data transfers we need to compare S_1 and S_2, which amounts to comparing the following quantities: $2 \times \texttt{size}(\mu(R, \varphi))$ and $2 \times \texttt{size}(\mu(F(R), \varphi)) + \texttt{size}(A) + \texttt{size}(R)$. In other words, we need to know whether the data removed from the fixpoint result (by pushing the filter into it) makes up for the sizes of A and R that are transferred to make the additional cogroup. It is clear that the result of this comparison depends on the data. So, with the help of a cost model with statistical information about the data, we could decide whether the rule is optimising data transfers. In the presence of such a cost model, we can refine the previously described process by making it perform the second optimisation only when judged cost-effective.

4 RELATED WORKS

Big data frameworks and extensions. Big data frameworks have gained much attention as general platforms for large-scale data intensive computations, propelled by a rather standard set of functional programming primitives to manipulate distributed collections of data. They now gain momentum as target languages for the compilation and distributed execution of domain

specific languages. In particular, the Spark framework [Zaharia et al. 2016] has been extended with higher-level programming constructs for: graph analytics [Gonzalez et al. 2014], machine learning pipelines [Meng et al. 2016], relational queries [Armbrust et al. 2015], and other domain-specific query languages such as Datalog [Shkapsky et al. 2016] and DIQL [Fegaras and Noor 2018]. As big data frameworks are increasingly considered as compiler backends, appropriate foundations for developing optimizing compilers become instrumental.

Algebraic approach. The research on algebraic foundations for the development of query optimizers can be roughly divided into two main lines of works. The first line of works gathers the seminal relational algebra [Codd 1970] and its numerous extensions. The second line of works gathers works that consolidate around the concepts of monad comprehensions [Wadler 1992] that evolved into ringad comprehensions [Gibbons 2016] and monoid algebras [Fegaras 2017; Tannen et al. 1991a,b]. The present work, that further builds on monoid algebras, belongs to the latter. Although the two lines of works exhibit many similarities – as pointed out in [Meijer and Bierman 2011], we argue that each comes with particular advantages and limitations in the context of distributed data collections.

Works based on Relational Algebra (RA). Compared to μ-monoids (and more generally to the second line of works), RA provides a very-well characterized and mature foundation, at the price of a formal model (relational tables) which is sometimes more rigid than it should for big data collections. In particular RA imposes: (1) to split data into columns to benefit from the theory (which often introduces expensive joins), (2) to be heavily dependent on user-defined functions (UDFs) that are supported (which limits the extensibility to heterogeneous workloads), (3) no built-in support for the distribution of data and computations, and (4) limited support for recursion: RA is notoriously known to be difficult to extend to support recursion, despite many attempts for optimizing recursive queries and iterative workloads over the last decades. Developing RA extensions that overcome one or more of these limitations is challenging.

The optimization of relational queries for distributed evaluation was tackled in SparkSQL [Armbrust et al. 2015]. Specific APIs (Dataframes and Datasets) were introduced in order to match columns in the relational world and expose a subset of SQL. The Catalyst framework [Armbrust et al. 2015] then compiles and optimizes these queries into physical plans executed by Spark. However SparkSQL lacks recursion operators, and the Catalyst optimizer supports only non-recursive plans.

The BigDatalog system [Shkapsky et al. 2016] introduces parallel semi-naive evaluation (PSN) as a distributed execution framework for recursive predicates. BigDatalog makes it possible to write recursive queries in a Datalog-style notation with two types of rules: exit rules and recursive rules. A recursive operator is introduced as a special driver operator that runs on the master and executes the PSN. The recursive operator is associated with two physical subplans: the plan for exit rules, and the plan for recursive rules. PSN first evaluates the exit rules plan and then repeatedly evaluates the recursive rules plan until a fixpoint is reached. This makes it possible to implement PSN using SetRDDs [Shkapsky et al. 2016], an improved version of Spark's distributed resilient datasets (RDDs) [Zaharia et al. 2016]. The approach shows significant performance gains in practice. However, the recursive operator is not integrated into the algebra (i.e. it is not fully compositional with other constructs). Instead, the two subplans associated with each recursive operator are optimized independently, and substituted in the PSN evaluation strategy. This greatly limits the room for more global optimizations since this makes recursive terms less amenable to general term transformations.

In contrast, extensions of RA that introduce an explicit recursive operator exposed at the algebraic level [Agrawal 1988; Jachiet et al. 2018], open the way to more general term transformations and optimizations. For instance, RA is extended with an alpha recursive operator in [Agrawal 1988], and a more general fixpoint operator in [Jachiet et al. 2018]. Both works investigate transformations of recursive terms (in the non-distributed setting). In [Jachiet et al. 2018] recursion is exposed at the algebraic level in a fully compositional way with other RA operators. Rewriting rules are then proposed, including pushing filters and joins through recursive terms, and even merging two recursive terms. Rewritings use colunm names to track whether a column has been removed or joined with another dataset or filtered upon. For example, a join of two recursive operators can, under certain conditions, be transformed into a single recursive operator, yielding significant practical performance improvements [Jachiet et al. 2018]. However, these works do not consider the distributed evaluation of recursive terms.

Compared to all RA-based works, the present work μ-monoids introduces a fixpoint operator in a monoid algebra, which is natively prone to express distributed optimizations. We proposed a way to evaluate this fixpoint operator in order to reduce the size of data transfers across the network. Our extended algebra μ-monoids is fully compositional. Recursive terms can contain `flatmap` operators with arbitrary UDFs, and where data is organised in a more flexible way than with RA. More specifically, μ-monoids allows working with data in its native format without having to flatten it or to convert it. In monoid algebra, outer collections and inner data inside those collections can be operated on using the same algebraic operations (we could for example have a `flatmap`, a `reduce` or a `groupby` over a list that is inside elements of an outer collection). In RA, it cannot be done this way because operators are defined for relations only (and not arbitrary lists inside a column). A similar possibility could be obtained by extending the relational algebra with a `map` operator with UDFs that, however, are not part of the algebra. The capacity of μ-monoids to treat nested data with the same algebraic constructions offers significant advantages. First, it allows designing nested query languages (with queries containing other queries querying over inner data) and easily translating them to the algebra. This was already shown by the DIQL query language [Fegaras and Noor 2018]. Second, it allows the algebra to capture a more holistic view of the data treatment and potentially get a better chance at optimizing this entire treatment. In comparison, in RA, UDFs are written in a language that is different from the algebra, which makes it harder to reason about and to optimise across UDFs.

Monoid algebras. μ-monoids builds on a line of previous works on monoid algebras, at the intersection between functional programming, type theory, and data management. The idea of using monoids and monoid homomorphisms for modeling computations with data collections originates from the works found in [Tannen et al. 1991a,b]. It was showed in [Buneman et al. 1995] that monads can be used to generalize nested relational algebra to different collection types and complex data. Wadler also explored and developed monad comprehensions [Wadler 1992] and ringad comprehensions [Gibbons 2016], inspired by the early works on list comprehensions [Peyton Jones 1987; Turner 2016]. These ideas were used in LINQ [Meijer et al. 2006] and Emma [Alexandrov et al. 2016] which are both comprehensions-based programming languages that facilitate the execution of entire programs by database systems. To be evaluated, LINQ translates into an intermediate form that can be executed on relational database systems supporting SQL:1999 [Meijer et al. 2006][4]. Emma [Alexandrov et al. 2016] is a comprehensions-based language similar in spirit, but which

[4]One particularity of SQL:1999 is that it adds the support of recursive queries through a "`with recursive`" construct in the query language. These recursive terms can be evaluated by mainstream relational engines such as PostgreSQL but they are not optimized, as observed in [Bagan et al. 2017; Jachiet et al. 2018]

rather targets JVM-based parallel dataflow engines (such as Spark and Flink), and without support for recursion. Fegaras proposed a monoid comprehension calculus [Fegaras and Maier 2000] which later evolved in the monoid algebra presented in [Fegaras 2017].

The present work on μ-monoids extends the monoid algebra of [Fegaras 2017] with a "μ" fixpoint operator. The algebra of [Fegaras 2017] has a `repeat` operator, however no optimization is provided for this operator. The major interest of the "μ" fixpoint operator of μ-monoids is that, under reasonable conditions (detailed in Section 2.5), 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. Furthermore, we propose rewriting rules to optimise expressions containing the fixpoint operator, including pushing filters through fixpoints and prefiltering a fixpoint before a join.

5 CONCLUSION

We propose a monoid algebra extended with a fixpoint operator that models recursion. This algebra is suitable for modeling computations with distributed data collections such as the ones found in big data frameworks. We propose a way to evaluate the fixpoint in a distributed manner. We propose rewriting rules for optimizing 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. 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 filter a fixpoint before a join. This makes it possible to develop algebraic optimizations in the presence of recursion, in a way which is especially suited for query optimizers and compilers targeting big data frameworks.

This opens several perspectives for further works. One perspective would be to extend pattern expressions with choice patterns (patterns with case classes and switch case conditions). Other perspectives would be to use the filter pushing technique to push filters into more generic expressions other than the fixpoint, and to explore when a join can be completely pushed inside the fixpoint, in particular by a refined analysis of φ subexpressions.

REFERENCES

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

Alexander Alexandrov, Asterios Katsifodimos, Georgi Krastev, and Volker Markl. 2016. Implicit Parallelism through Deep Language Embedding. *SIGMOD Record* 45, 1 (2016), 51–58. https://doi.org/10.1145/2949741.2949754

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. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. 1383–1394. https://doi.org/10.1145/2723372.2742797

Guillaume Bagan, Angela Bonifati, Radu Ciucanu, George H. L. Fletcher, Aurélien Lemay, and Nicky Advokaat. 2017. gMark: Schema-Driven Generation of Graphs and Queries. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2017), 856–869. https://doi.org/10.1109/TKDE.2016.2633993

Peter Buneman, Shamim A. Naqvi, Val Tannen, and Limsoon Wong. 1995. Principles of Programming with Complex Objects and Collection Types. *Theor. Comput. Sci.* 149, 1 (1995), 3–48. https://doi.org/10.1016/0304-3975(95)00024-Q

Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4 (2015), 28–38. http://sites.computer.org/debull/A15dec/p28.pdf

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

Leonidas Fegaras. 2017. An algebra for distributed Big Data analytics. *Journal of Functional Programming* 27 (2017), e27. https://doi.org/10.1017/S0956796817000193

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

Leonidas Fegaras and Md Hasanuzzaman Noor. 2018. Compile-Time Code Generation for Embedded Data-Intensive Query Languages. In *2018 IEEE International Congress on Big Data, BigData Congress 2018, San Francisco, CA, USA, July 2-7, 2018.* 1–8. https://doi.org/10.1109/BigDataCongress.2018.00008

Jeremy Gibbons. 2016. 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 (Lecture Notes in Computer Science)*, Vol. 9600. Springer, 132–151. https://doi.org/10.1007/978-3-319-30936-1_7

Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* 599–613. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez

Louis Jachiet, Nils Gesbert, Pierre Genevès, and Nabil Layaïda. 2018. On the Optimization of Recursive Relational Queries. In *BDA 2018 - 34ème Conférence sur la Gestion de Données - Principes, Technologies et Applications.* Bucarest, Romania, 1–22. https://hal.inria.fr/hal-01673025

Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. LINQ: reconciling object, relations and XML in the .NET framework. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006.* 706. https://doi.org/10.1145/1142473.1142552

Erik Meijer and Gavin Bierman. 2011. A Co-relational Model of Data for Large Shared Data Banks. *Commun. ACM* 54, 4 (April 2011), 49–58. https://doi.org/10.1145/1924421.1924436

Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17 (2016), 34:1–34:7. http://jmlr.org/papers/v17/15-237.html

Shoumik Palkar, James J. Thomas, Deepak Narayanan, Pratiksha Thaker, Rahul Palamuttam, Parimarjan Negi, Anil Shanbhag, Malte Schwarzkopf, Holger Pirk, Saman P. Amarasinghe, Samuel Madden, and Matei Zaharia. 2018. Evaluating End-to-End Optimization for Data Analytics Applications in Weld. *PVLDB* 11, 9 (2018), 1002–1015. https://doi.org/10.14778/3213880.3213890

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

Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. 2016. Big Data Analytics with Datalog Queries on Spark. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* 1135–1149. https://doi.org/10.1145/2882903.2915229

Val Tannen, Peter Buneman, and Shamim A. Naqvi. 1991a. 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.* 9–19.

Val Tannen, Peter Buneman, and Atsushi Ohori. 1991b. Data Structures and Data Types for Object-Oriented Databases. *IEEE Data Eng. Bull.* 14, 2 (1991), 23–27. http://sites.computer.org/debull/91JUN-CD.pdf

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

Philip Wadler. 1992. Comprehending Monads. *Mathematical Structures in Computer Science* 2, 4 (1992), 461–493. https://doi.org/10.1017/S0960129500001560

Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan R. Sparks, Aaron Staple, and Matei Zaharia. 2016. Matrix Computations and Optimization in Apache Spark. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016.* 31–38. https://doi.org/10.1145/2939672.2939675

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. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

A ADDITIONAL PROOFS

We have $\psi = R \uplus \varphi$ and we suppose (fc): $\forall A, B \quad \varphi(A \uplus B) = \varphi(A) \uplus \varphi(B)$

A.1 Proving the existence of the fixpoint

We prove that $\mu(\psi)$, the fixpoint of $f : X \to X \cup \psi(X)$ exists:

We consider the following bag $F = \biguplus_{n\in\mathbb{N}} \varphi^{(n)}(R)$

We have $\varphi(F) = \varphi(\biguplus_{n\in\mathbb{N}} \varphi^{(n)}(R)) = \biguplus_{n\in\mathbb{N}} \varphi^{(n+1)}(R)$ (fc)

So $\varphi(F) = \biguplus_{n\in\mathbb{N}^*} \varphi^{(n)}(R)$

So $\psi(F) = R \uplus \varphi(F) = R \uplus \biguplus_{n\in\mathbb{N}^*} \varphi^{(n)}(R) = F$ (because $\varphi^{(0)}(R) = R$)

So $f(F) = F \cup \psi(F) = F \cup F = D_f$, where $D_f = \text{distinct}(F)$

We can find bags $F_1, ..., F_n$, all without duplicates such that $F = D_f \uplus F_1 \uplus ... \uplus F_n$ (they can be the singletons remaining after removing D_f from F)

We have $f(F) = F \cup \psi(F) = F \cup (R \uplus \varphi(F)) = F \cup R \cup \varphi(D_f \uplus F_1 \uplus ... \uplus F_n) = F \cup R \cup (\varphi(D_f) \uplus \varphi(F_1) \uplus ... \uplus \varphi(F_n))$ (fc)

So $f(F) = F \cup R \cup \varphi(D_f) \cup \varphi(F_1) \cup ... \cup \varphi(F_n)$

We have $\forall i, F_i \subset D_f$ (because F_i do not contain duplicates and are contained in F), so $\varphi(F_i) \subset \varphi(D_f)$ (fc)

Which means $f(F) = F \cup R \cup \varphi(D_f) = D_f \cup R \cup \varphi(D_f)$ (because $\cup$ removes duplicates from bag union)

Since $f(F) = D_f$ (as shown earlier), we have $D_f \cup R \cup \varphi(D_f) = D_f$, which means $f(D_f) = D_f$

Hence f has a fixpoint $\mu(\psi) = D_f = \text{distinct}(\biguplus_{n\in\mathbb{N}} \varphi^{(n)}(R)) = \bigcup_{n\in\mathbb{N}} \varphi^{(n)}(R)$

A.1.1 More properties of fixpoint expressions. We have $\mu(R \uplus \varphi) = \bigcup_{n\in\mathbb{N}} \varphi^{(n)}(R) = \bigcup_{n\in\mathbb{N}} \varphi(\biguplus_{r\in R}\{r\}) = \bigcup_{n\in\mathbb{N}}(\biguplus_{r\in R} \varphi(\{r\})) = \bigcup_{n\in\mathbb{N}}(\bigcup_{r\in R} \varphi(\{r\}))$

Which means that $\forall a \in \mu(R \uplus \varphi)\ a \in \varphi^{(n)}(\{r\})$ for some $r \in R$ and $n \in N$
Also, $a \in \text{distinct}(\varphi^{(n)}(\{r\}))$ and $\forall n \in \mathbb{N}\ \forall r \in R\ \text{distinct}(\varphi^{(n)}(\{r\})) \subset \mu(\varphi)$

A.2 Proving that join is a homomorphism $\uplus \to \uplus$

join is a homomorphism $\uplus \to \uplus$

We can verify from the formula of the join expressions that $join(X, A)$ is the bag $\{(k, (x, a)) \mid (k, x) \in X, (k, a) \in A\}$ (see A.4)

$join(X_1 \uplus X_2, A) = \{(k, (x, a)) \mid (k, x) \in X_1 \uplus X_2, (k, a) \in A\} = \{(k, (x, a)) \mid (k, x) \in X_1, (k, a) \in A\} \uplus \{(k, (x, a)) \mid (k, x) \in X_2, (k, a) \in A\}$

A.3 Proving that the fixpoint operator is a homomorphism from $\uplus$ to $\cup$

$\forall R_1, R_2 \quad \mu(R_1 \uplus R_2, \varphi) = \mu(R_1, \varphi) \cup \mu(R_2, \varphi)$:

We put $F = \mu(R_1 \uplus R_2, \varphi)$

$$
\begin{aligned}
F &= \bigcup_{n\in\mathbb{N}} \varphi^{(n)}(R_1 \uplus R_2) \\
&= \bigcup_{n\in\mathbb{N}} (\varphi^{(n)}(R_1) \uplus \varphi^{(n)}(R_2)) \text{ (because } \varphi \text{ satisfies (fc))} \\
&= (\bigcup_{n\in\mathbb{N}} \varphi^{(n)}(R_1)) \cup (\bigcup_{n\in\mathbb{N}} \varphi^{(n)}(R_2)) = \mu(R_1, \varphi) \cup \mu(R_2, \varphi)
\end{aligned}
$$

A.4 Bag formula for join

We have join(X,Y) = $\texttt{flatmap}(\lambda(k,(s_x,s_y)).\ \texttt{flatmap}(\lambda x.\ \texttt{flatmap}(\lambda y.\ \{(k,(x,y))\}, s_y), s_x), \texttt{cogroup}(X,Y))$

We put $A = join(X,Y)$ and show that $A = B$, where $B = \{(k,(x,y)) \mid (k,x) \in X, (k,y) \in Y\}$

Following the bag definition of flatmap, we have $A = \biguplus_{(k,(s_x,s_y))\in \texttt{cogroup}(X,Y)} \biguplus_{x\in s_x} \biguplus_{y\in s_y} \{(k,(x,y))\}$

Let $e \in A$

$\exists (k,(s_x,s_y)) \in \texttt{cogroup}(X,Y)$ such that $\exists x \in s_x\ \exists y \in s_y \quad e = (k,(x,y))$

Since $(k,(s_x,s_y)) \in \texttt{cogroup}(X,Y)$, $s_x = \{v \mid (k,v) \in X\}$ (from the definition of cogroup), and since $x \in s_x$, then $(k,x) \in X$

Similarly, $(k,y) \in Y$

So $e = (k,(x,y)) \in B$

Let $(k,(x,y)) \in B$

We then have $(k,x) \in A$ and $(k,y) \in B$. So $k \in keys(A) \cup keys(B)$

Let $s_x = \{v \mid (k,x) \in X\}$ and $s_y = \{v \mid (k,y) \in Y\}$

So $(k,(s_x,s_y)) \in \texttt{cogroup}(A,B)$ and $x \in s_x$, $y \in s_y$, so $(k,(x,y)) \in A$