An Algebra for Distributed Big Data Analytics

Leonidas Fegaras
University of Texas at Arlington
(*e-mail:* fegaras@cse.uta.edu)

Abstract

We present an algebra for data-intensive scalable computing based on monoid homomorphisms that consists of a small set of operations that capture most features supported by current domain-specific languages for data-centric distributed computing. This algebra is being used as the formal basis of MRQL, which is a query processing and optimization system for large-scale distributed data analysis. The MRQL semantics is given in terms of monoid comprehensions, which support group-by and order-by syntax and can work on heterogeneous collections without requiring any extension to the monoid algebra. We present the syntax and semantics of monoid comprehensions and provide rules to translate them to the monoid algebra. We give evidence of the effectiveness of our algebra by presenting some important optimization rules, such as converting nested queries to joins.

1 Introduction

New frameworks in distributed processing have become essential tools to large-scale data analysis. Among these frameworks, the Map-Reduce programming model (Dean and Ghemawat, 2004) was one of the first to emerge as a generic, scalable, and cost effective solution for Big Data processing on clusters of commodity hardware. The Map-Reduce framework was inspired by functional programming. For each Map-Reduce job, one needs to provide two functions: a map and a reduce. The map function specifies how to process a single key-value pair to generate a set of intermediate key-value pairs, while the reduce function specifies how to combine all intermediate values associated with the same intermediate key. The Map-Reduce framework uses the map function to process the input key-value pairs in parallel by partitioning the data across a number of compute nodes in a cluster (the map workers), which execute the map task in parallel without communicating with each other. Then, the map results are repartitioned (shuffled) across a number of compute nodes (the reduce workers) so that values associated with the same key are grouped and processed by the same reduce worker. Finally, each reduce worker applies the reduce function to every group in its assigned partition and stores the job results into the distributed file system.

But, very soon, it became apparent that, because of its simplicity, the Map-Reduce model has many limitations. One of its major drawbacks is that, to simplify reliability and fault tolerance, it stores the intermediate results between the Map and Reduce stages and between consecutive Map-Reduce jobs on secondary storage, rather than in memory, which imposes a high overhead to complex workflows and graph algorithms. This drawback makes this framework ill-suited for certain Big Data workloads, such as real-time analytics, continuous queries, and iterative algorithms. To address the inherent limitations of the Map-Reduce model, new alternative frameworks have been introduced recently that perform better for a wider spectrum of workloads. Currently, among them, the most promising frameworks that seem to be good alternatives to Map-Reduce while addressing its drawbacks are Google's Pregel (Malewicz *et al.*, 2010), Apache Spark (Apache Spark, 2017), and Apache Flink (Apache Flink, 2017), which are in-memory distributed computing systems. We collectively refer to all these data-intensive distributed computing environments as DISC (Data-Intensive Scalable Computing) programming environments (Bryant, 2011). Some of these DISC programming environments, notably Spark and Flink, provide a functional-style API that consists of higher-order operations, similar to those found in functional programming languages. By adopting a functional programming style, not only do these frameworks prevent interference among parallel tasks, but they also facilitate a functional style in composing complex data analysis computations using powerful higher-order operations as building blocks. These benefits have already been explored and capitalized in earlier data parallel languages, such as NESL (Blelloch, 1993), and in database languages, such as LINQ and XQuery.

Even though, in principle, these DISC frameworks provide APIs that are simple to understand, it is hard to develop non-trivial applications coded in a general-purpose programming language. In addition, there are many configuration parameters to adjust for better performance that overwhelm non-expert users. Because of the complexity involved in developing and fine-tuning data analysis applications using the provided APIs, most programmers prefer to use declarative languages, such as Apache Hive (Apache Hive, 2017) and Pig (Olston *et al.*, 2008), to code their distributed applications, instead of coding them directly in an algorithmic language. For instance, based on data from few years back, Hive was used for over 90% of Facebook Map-Reduce jobs and Pig Latin was used for roughly 1/3 of all Yahoo! Map-Reduce jobs. There are many reasons why programmers prefer domain-specific declarative languages. First, it is hard to develop, optimize, and maintain non-trivial applications coded in a non-declarative language. Second, given the multitude of the new emerging DISC frameworks, such as Spark and Flink, it is hard to tell which one of them will prevail in the near future. Data intensive applications that have been coded in one of these paradigms may have to be rewritten as technologies evolve. Hence, it would be highly desirable to express these applications in a declarative language that is independent of the underlying distributed platform. Furthermore, the evaluation of such applications can benefit from cost-based optimizations and automatic parallelism provided by the DSL-based systems, thus relieving the application developers from the intricacies of Big Data analytics and distributed computing.

But what will be a good data model, algebra, and domain-specific language for DISC applications? Data parallelism is achieved when each processor performs the same task on different pieces of distributed data. This means that the task results should not depend on how we divide the data among processors. Hence, if we want to allow processors to work on groups of data in parallel, we would need to use associative operations. Therefore, associativity is essential for data parallelism. By using associative operations, the intermediate values can be grouped arbitrarily or even aggregated in stages. This was evident even in the early data parallel languages, such as NESL (Blelloch and Sabot, 1990).

A second observation is that a data model for data-centric distributed processing must support both lists and bags (multisets). It needs to support lists because order of data is important to some applications, such as for scientific applications that work on vectors and matrices. But it also needs to support bags to take advantage of the multitude of high-performance algorithms that can only apply when the order of the result is insignificant. Without bags, for example, nested loops on collections can not be evaluated using joins, since a join may return the results in a different order than the nested loop. By excluding joins, we restrict ourselves to suboptimal evaluations. This observation too is evident in practical relational query languages, such as SQL, for which the main collection type is a bag, not a list.

We present an algebra for data-intensive distributed processing based on monoid homomorphisms. Monoids and monoid homomorphisms directly capture the most important property required for data parallelism, namely associativity. They fully support the functionality provided by current domain-specific languages for data-centric distributed processing by directly supporting operations, such as group-by, order-by, aggregation, and joins between heterogeneous collections. Formal frameworks based on collection monads, such as monad comprehensions (Wadler, 1990), on the other hand, require special extensions (such as group-by, order-by, aggregation, and monad morphisms) to achieve the same expressiveness (Gibbons, 2016). We believe that extending collection monads with all these additional operations as well as with the laws that these operations need to obey would unnecessarily complicate optimization. Monads and monad comprehensions have been shown to be very valuable tools for many programming languages, most notably Haskell. Supporting a domain-specific syntax in the form of monad comprehensions in an existing functional programming language makes easier to express data-centric queries deeply embedded in the language. This is a great idea for languages that already support monads and monad comprehensions, because computations on data collections can be combined with other monads. In this paper, though, we claim that collection monoids and monoid homomorphisms are actually a better formal basis for DISC query languages since they do not need any extensions to effectively support the functionality of these languages.

The focus of our work is on the design of an effective algebra that can express most features found in many current domain-specific languages for data-centric distributed computing and, at the same time, can be easily translated to a wide spectrum of efficient distributed operations, taking full advantage of the functionality provided by the underlying DISC platform. This algebra is the formal basis of MRQL (Apache MRQL, 2017), which is a query processing and optimization system for large-scale, distributed data analysis. It is currently an Apache incubating project with many developers and users worldwide.

The contributions of this paper can be summarized as follows:

- We provide a formal framework for the monoid algebra based on collection monoids and collection homomorphisms (Section 3).
- We present a novel algebra for distributed computing based on collection homomorphisms, called the monoid algebra, which consists of a small set of operations that capture most features supported by current domain-specific languages for data-intensive distributed computing (Section 4).

- We give evidence on the effectiveness of the monoid algebra for expressing distributed computations by providing a program expressed in this algebra that simulates general Bulk Synchronous Parallel (BSP) computations efficiently, requiring the same amount of data shuffling as a typical BSP implementation (Section 5).
- We compare the monoid algebra with algebras based on monads and we show that the monoid algebra does not require any extension in the form of added non-homomorphic operations to capture the functionality required by practical DISC query languages, while algebras based on monads require various extensions that must obey additional laws (Section 6).
- We present the syntax and semantics of monoid comprehensions and provide rules to translate them to the monoid algebra (Section 7).
- We describe the MRQL syntax and provide rules to translate it to monoid comprehensions (Section 8).
- To support our claim that the monoid algebra facilitates query optimization, we present transformations for converting nested queries to joins (Section 9) and for translating self-joins to group-bys (Section 10).
- We show the effectiveness of our optimizations through experiments on three data analysis queries: PageRank, a nested join query, and k-means clustering (Section 11).
- Finally, we show how to use the monoid algebra to convert batch data-analysis queries to incremental stream processing programs (Section 12).

2 Related Work

Database and data-intensive distributed processing share many common goals and motivations as they both need to work on large collections of data. In the 90's, there was a need for new algebras and query languages that go beyond the limitations of the relational model to capture the emerging object-oriented databases. During that time, there was a prolific collaboration between the programming language and database communities to develop formal frameworks and database query languages that can handle complex objects and nested collections. One of the first data models considered was based on monoids and monoid homomorphisms (Tannen *et al.*, 1991). Later, based on this theory, a monoid comprehension calculus was introduced and used as a formal basis for ODMG OQL (Fegaras and Maier, 1995; Fegaras and Maier, 2000). In the meantime, there was a substantial body of work on extending monads and monad comprehensions to be used as a formal basis for the emerging database query languages. Monad comprehensions were first introduced by Wadler (Wadler, 1990) as a generalization of list comprehensions. They were first proposed as a convenient database language by Trinder and Wadler (Trinder and Wadler, 1991; Trinder, 1991), who also presented algebraic transformations over these forms as well as methods for converting comprehensions into joins. These monad comprehensions, which are over data collections, such as lists, bags, and sets, are based on collection monads (also called ringads (Gibbons, 2016)), which are monads extended with an empty collection and a merge function. The monad comprehension syntax was also adopted by Buneman, et al. (Buneman *et al.*, 1994) as an alternative syntax to monoid homomorphisms. The monad comprehension syntax was used for capturing operations that involve collections of the same type, while structural recursion was used for expressing the rest of the operations, such as converting one collection type to another and aggregations. One important extension towards bringing comprehensions closer to a practical query language was the work on list comprehensions with group-by and order-by (Wadler and Peyton Jones, 2007).

Our monoid algebra described in Section 4 extends our earlier work (Fegaras and Maier, 1995; Fegaras and Maier, 2000) on monoid homomorphisms by supporting three new operations that are very important for DISC frameworks: groupBy, orderBy, and coGroup, which are expressed as monoid homomorphisms. It also extends our earlier work on monoid comprehensions by providing declarative syntax for expressing group-by and order-by operations in a comprehension. All these extensions are captured as monoid homomorphisms. Section 6 gives a detailed comparison between monoid comprehensions and monad comprehensions.

Monad comprehensions were generalized in (Grust and Scholl, 1999) to work on mixed collections and to capture aggregation. The core primitive for data processing in (Grust and Scholl, 1999) is the foldr operation, which is structural recursion over the insert-representation of collections types. The generalized comprehensions in (Grust and Scholl, 1999) are equivalent to monoid comprehensions, even though monoid comprehensions are based on structural recursion on the union-representation of collections. Furthermore, the well-definedness conditions for the left-commutative and left-idempotent properties of foldr are equivalent to the commutativity and idempotence conditions for "well-formed" homomorphisms in (Fegaras and Maier, 1995). Unlike monoid homomorphisms, foldr and foldl computations are hard to parallelize (Steele, 2009), although there are methods for deriving cost-optimal list homomorphisms from a pair of foldr and foldl based on the third homomorphism theorem (Morita *et al.*, 2007).

DISC systems are data-parallel systems on clusters of shared-nothing computers connected through a high-speed network. One of the earliest DISC frameworks was the Map-Reduce model, which was introduced by Google in 2004 (Dean and Ghemawat, 2004). The most popular Map-Reduce implementation is Apache Hadoop (Apache Hadoop, 2017), an open-source project developed by Apache, which is used today by many companies to perform data analysis. There are also a number of higher-level languages that make Map-Reduce programming easier, such as HiveQL (Thusoo *et al.*, 2009), PigLatin (Olston *et al.*, 2008), SCOPE (Chaiken *et al.*, 2008), and Dryad/Linq (Isard and Yu, 2007). Apache Hive (Thusoo *et al.*, 2009; Thusoo *et al.*, 2010) provides a logical RDBMS environment on top of the Map-Reduce engine, well-suited for data warehousing. Using its high-level query language, HiveQL, users can write declarative queries, which are optimized and translated into Map-Reduce jobs that are executed using Hadoop. HiveQL does not handle nested collections uniformly: it uses SQL-like syntax for querying data sets but uses vector indexing for nested collections. Unlike MRQL, HiveQL has many limitations. It does not allow query nesting in predicates and select expressions, but allows a table reference in the from-part of a query to be the result of a select-query. Because of these limitations, HiveQL enables users to plug-in custom Map-Reduce scripts into queries. Although Hive uses simple rule-based optimizations to translate queries, it has yet to provide a comprehensive framework for cost-based optimizations. Apache Pig (Gates *et al.*, 2009) resembles Hive as it provides a user-friendly scripting language, called PigLatin (Olston *et al.*, 2008), on top of Map-Reduce, which allows explicit filtering, map, join, and group-by operations.

Like Hive, PigLatin performs very few optimizations based on simple rule transformations. PACT/Nephele (Battre *et al.*, 2010) is a Map-Reduce programming framework based on workflows, which consist of high-order operators, such as map and reduce. These workflows are converted to logical execution plans for Nephele, a general distributed program execution engine. Even though PACT/Nephele workflow programs are very flexible and are not limited to rigid Map-Reduce pairs, they are hard to program, since programmers have to construct low-level workflows. SCOPE (Chaiken *et al.*, 2008), an SQL-like scripting language for large-scale analysis, does not support sub-queries but provides syntax to simulate sub-queries using outer-joins. Like Hive, because of its limitations, SCOPE provides syntax for user-defined process/reduce/combine operations to capture explicit Map-Reduce computations.

Recent DISC systems go beyond Map-Reduce by maintaining dataset partitions in the memory of the compute nodes. These systems include the main memory Map-Reduce M3R (Shinnar *et al.*, 2012), Apache Spark (Apache Spark, 2017), Apache Flink (Apache Flink, 2017), Piccolo (Power and Li, 2010), and distributed GraphLab (Low *et al.*, 2012). Another alternative framework to the Map-Reduce model is the Bulk Synchronous Parallelism (BSP) programming model (Valiant, 1990). The best known implementations of the BSP model for Big Data analysis are Google's Pregel (Malewicz *et al.*, 2010), Apache Giraph (Apache Giraph, 2017), and Apache Hama (Apache Hama, 2017).

One of the advantages of using DSL-based systems, such as Pig, Hive, and MRQL, to develop DISC applications is that these systems support automatic program optimization. Such program optimization is harder to do in an API-based system. Some API-based systems though have found ways to circumvent this shortcoming. The evaluation of RDD (Resilient Distributed Dataset) transformations in Spark, for example, is deferred until an action is encountered that brings data to the master node or stores the data into a file. Spark collects the deferred transformations into a DAG and divides them into subsequences, called stages, which are similar to Pig's Map-Reduce barriers. Data shuffling occurs between stages while transformations within a stage are combined into a single RDD transformation. Unlike Pig though, Spark cannot perform non-trivial optimizations, such as moving a filter operation before a join, because the functional arguments used in RDD operations are written in the host language and cannot be analyzed for code patterns at run-time. Spark has addressed this shortcoming by providing two additional APIs, called DataFrames and Datasets (Armbrust *et al.*, 2015). A Dataset combines the benefits of RDD (strong typing and powerful higher-order operations) with Spark SQL's optimized execution engine. A DataFrame is a Dataset organized into named columns as in a relational table. SQL queries in Datasets and DataFrames are translated and optimized into RDD workflows at run-time.

The closest work to ours is Emma (Alexandrov *et al.*, 2016), which is a language for parallel data analysis that is deeply embedded in Scala. Unlike our work, Emma does not provide an SQL-like query syntax; instead, it uses Scala's for-comprehensions as the core language abstraction to query datasets. These for-comprehensions are optimized and translated to abstract dataflows at compile-time, and these dataflows are evaluated at run-time using just-in-time code generation. Using the host language syntax for querying allows a deeper embedding of DSL code into the host language but it requires that the host language supports meta-programming and provides a declarative syntax for querying collections, such as for-comprehensions. Furthermore, Scala's for-comprehensions do not provide a declarative syntax for group-by. Emma's core primitive for data processing is the fold operation over the union-representation of bags, which is equivalent to a bag homomorphism. The fold well-definedness conditions are similar to the preconditions for bag homomorphisms, as described in (Fegaras and Maier, 1995). Scala's for-comprehensions are translated to monad comprehensions, which are desugared to monad operations, which, in turn, are expressed in terms of fold. Non-monadic operations, such as aggregations, are expressed as folds. Emma also provides additional operations, such as groupBy and join, but does not provide algebraic operations for sorting, outer-join (needed for non-trivial query unnesting), and repetition (needed for iterative workflows). Unlike MRQL, Emma does not provide general methods to convert nested correlated queries to joins, except for simple nested queries in which the domain of a generator qualifier is another comprehension. Another API-based distributed system that supports run-time optimization is Summingbird (Boykin *et al.*, 2014), which can run on both Map-Reduce and Storm. Compared to Spark and Flink APIs, the Summingbird API is intentionally restrictive to facilitate optimizations at run-time. It supports map, flatMap, filter, sumByKey, and join. The sumByKey operation, which is similar to Spark's reduceByKey, is a group-by followed by a reduction that aggregates the grouped values using a semigroup (an associative binary operation). Although the semigroup restriction excludes incorrect programs, Summingbird does not address the main shortcoming of the API-based approaches, which is their inability to analyze the functional arguments of the map-like computations at run-time to do complex optimizations.

3 Collection Monoids and Collection Homomorphisms

The monoid algebra consists of a small number of higher-order operators that can be expressed as monoid homomorphisms defined using structural recursion based on the union representation of bags and on the append representation of lists (Fegaras and Maier, 1995; Fegaras and Maier, 2000).

In abstract algebra, a monoid is an algebraic structure equipped with a single associative binary operation and a single identity element. More formally, given a set S, a binary operator $\otimes$ from $S \times S$ to S, and an element $e \in S$, the structure $(S, \otimes, e)$ is called a monoid if $\otimes$ is associative and has an identity e:

$$\begin{array}{rcll} x \otimes (y \otimes z) & = & (x \otimes y) \otimes z & \quad \text{for all } x,y,z \in S \\ x \otimes e & = & x \;=\; e \otimes x & \quad \text{for all } x \in S \end{array}$$

Monoids may obey additional algebraic laws. The monoid $(S, \otimes, e)$ is commutative if $x \otimes y = y \otimes x$, for all $x, y \in S$. It is idempotent if $x \otimes x = x$, for all $x \in S$. For example, $(\mathbb{N}, +, 0)$ is a commutative monoid on natural numbers. Given that a monoid $(S, \otimes, e)$ can be identified by its operation $\otimes$, it is simply referred to as $\otimes$, with $\mathbf{1}_\otimes$ to denote its identity e and the type $\mathbb{T}_\otimes$ to denote the type of its carrier set S. Given two monoids $\otimes$ and $\oplus$, a monoid homomorphism H from $\otimes$ to $\oplus$ is a function from $\mathbb{T}_\otimes$ to $\mathbb{T}_\oplus$ that respects the monoid structure:

$$\begin{array}{rcll} H(X \otimes Y) & = & H(X) \oplus H(Y) & \quad \text{for all } X \text{ and } Y \text{ of type } \mathbb{T}_\otimes \\ H(\mathbf{1}_\otimes) & = & \mathbf{1}_\oplus & \end{array}$$

	$\circledast$	$\mathbb{T}_{\circledast}(\alpha)$	$\mathbf{1}_{\circledast}$	$\mathbb{U}_{\circledast}(x)$	additional laws
list	$++$	$[\alpha]$	$[\,]$	$[x]$	
bag	$\uplus$	$\{\!\{\alpha\}\!\}$	$\{\!\{\,\}\!\}$	$\{\!\{x\}\!\}$	commutativity
set	$\cup$	$\{\alpha\}$	$\{\,\}$	$\{x\}$	commutativity & idempotence

Fig. 1. Some Collection Monoids

Collection Monoids. We are interested in specifying monoid homomorphisms on data collections, such as lists, sets, and bags. Our treatment of collection types is based on the work of Aït-Kaci on collection monoids (Aït-Kaci, 2013, Appendix C). Let $\circledast$ be a binary operation that constructs a syntactic term from two other syntactic terms, starting from a base set S. A syntactic algebra is the set T^* of $\circledast$-terms on the set S, defined inductively as the limit of $\cup_{n\geq 0}T^n$, where:

$$T^n \;=\; \begin{cases} S & \text{if } n=0 \\ \{\, t_1 \circledast t_2 \mid t_1 \in T^i,\, t_2 \in T^j,\; i+j=n-1 \,\} & \text{if } n>0 \end{cases}$$

In other words, a term in the set T^* is a binary tree with $\circledast$ nodes and leaves from S. We now consider equivalence classes of terms in T^* under some laws, with the goal of defining collection monoids purely syntactically. These algebraic laws are the monoid laws: associativity, commutativity, and idempotence. These algebraic laws on the purely syntactic algebra define congruence classes, where each class contains equivalent terms under these algebraic laws. In other words, a syntactic monoid becomes a quotient monoid whose domain is a collection of equivalence classes that contain congruent terms modulo the algebraic laws. For example, the terms $1 \circledast (2 \circledast 3)$ and $(1 \circledast 2) \circledast 3$ from the syntactic monoid $\circledast$ over the base set $\mathbb{N}$ belong to the same congruence class. Although the T^* definition is a well-formed mathematical structure, it is not type-correct because it mixes elements of S with term constructions. To make it type-correct, we wrap the elements in S with a unit injection function $\mathbb{U}_{\circledast}$ of type $S \to T^*$. In addition, if we restrict the elements of the base set S to be of the same type t, we can represent a collection monoid as a parametric data type $\mathbb{T}_{\circledast}(t)$ that represents a collection of values of type t. We call this syntactic monoid a *collection monoid*. Fig. 1 shows some well-known collection types captured as collection monoids. For example, $\{\!\{1\}\!\} \uplus \{\!\{2\}\!\} \uplus \{\!\{1\}\!\}$ constructs the bag $\{\!\{1,2,1\}\!\}$.

Collection Homomorphisms. Let $\otimes$ be a collection monoid and $\oplus$ be a monoid that obeys all the laws of $\otimes$. A monoid homomorphism $\mathscr{H}(\oplus, f)$ from a collection monoid $\otimes$ to a monoid $\oplus$ is the homomorphic extension of f that satisfies $\mathscr{H}(\oplus, f) \circ \mathbb{U}_{\otimes} = f$. That is, for $H = \mathscr{H}(\oplus, f)$, we have the following definition:

$$H(X \otimes Y) \;=\; H(X) \oplus H(Y) \tag{1a}$$

$$H(\mathbb{U}_{\otimes}(x)) \;=\; f(x) \tag{1b}$$

$$H(\mathbf{1}_{\otimes}) \;=\; \mathbf{1}_{\oplus} \tag{1c}$$

For a function f of type $\alpha \to \mathbb{T}_{\oplus}$, the monoid homomorphism $\mathscr{H}(\oplus, f)$ maps a collection of type $\mathbb{T}_{\otimes}(\alpha)$ to a value of type $\mathbb{T}_{\oplus}$. In other words, $\mathscr{H}(\oplus, f)$ collects all the f-images of the elements of a collection of type $\mathbb{T}_{\otimes}(\alpha)$ using the $\oplus$ operation. For example, $\mathscr{H}(+, \lambda x.\, 1)\, X$ over the bag X returns the number of elements in X, while $\mathscr{H}(+, \lambda x.\, \{\!\{x\}\!\})\, L$ over the list L converts the list to a bag.

The requirement that $\oplus$ must obey all the laws of $\otimes$ excludes non-homomorphic functions on collection monoids. For example, the function bag2list that converts a bag to a list is not a homomorphism because, unlike bag union, list append is not commutative. If bag2list were a homomorphism, it would have led to the false statement:

$$\begin{aligned}&\text{bag2list}(X) \mathbin{+\!\!+} \text{bag2list}(Y) \;=\; \text{bag2list}(X \uplus Y) \;=\; \text{bag2list}(Y \uplus X)\\ &=\; \text{bag2list}(Y) \mathbin{+\!\!+} \text{bag2list}(X)\end{aligned}$$

which is false, because $\mathbin{+\!\!+}$ is not commutative. As another example, the function card that computes set cardinality, is not a homomorphism, because the monoid $+$ is not idempotent. If card were a homomorphism, it would have led to the false statement:

$$1 \;=\; \text{card}(\{x\}) \;=\; \text{card}(\{x\} \cup \{x\}) \;=\; \text{card}(\{x\}) + \text{card}(\{x\}) \;=\; 2$$

We use the partial order $\otimes \preceq \oplus$ between the monoids $\otimes$ and $\oplus$ to indicate that $\oplus$ obeys all the laws of $\otimes$. For example, $\mathbin{+\!\!+} \prec \uplus$ (i.e., $\mathbin{+\!\!+} \preceq \uplus$ but $\uplus \not\preceq \mathbin{+\!\!+}$), since $\uplus$ is commutative while $\mathbin{+\!\!+}$ is not.

Binary Homomorphisms. To capture binary equi-joins, we would need to define binary functions that are homomorphic on both inputs together. More specifically, we want to define a function H from the collection monoids $\oplus$ and $\otimes$ to the monoid $\odot$ such that:

$$H(X \oplus X', Y \otimes Y') = H(X,Y) \odot H(X',Y') \qquad \text{for all } X,\, X',\, Y, \text{ and } Y' \tag{2}$$

Note that H may not be homomorphic on each input separately because $H(X \oplus X', Y)$ is not necessarily equal to $H(X,Y) \odot H(X',Y)$. With some abuse of terminology, we call H a binary homomorphism. We now define a class of binary homomorphisms that can be expressed as the union of two collection homomorphisms. Let $\mathscr{H}(\odot, f_x)$ be a collection homomorphism from $\oplus$ to $\odot$ and $\mathscr{H}(\odot, f_y)$ be a collection homomorphism from $\otimes$ to $\odot$, such that:

$$f_x(x) \odot f_y(y) = f_y(y) \odot f_x(x) \qquad \text{for all } x \text{ and } y \tag{3}$$

A binary homomorphism $\mathscr{H}(\odot, f_x, f_y)$ from the collection monoids $\oplus$ and $\otimes$ to the monoid $\odot$ is defined as follows:

$$\mathscr{H}(\odot, f_x, f_y)(X,Y) \;\triangleq\; \mathscr{H}(\odot, f_x)X \;\odot\; \mathscr{H}(\odot, f_y)Y \qquad \text{for all } X \text{ and } Y \tag{4}$$

For a function f_x of type $\alpha \rightarrow \mathbb{T}_{\odot}$ and a function f_y of type $\beta \rightarrow \mathbb{T}_{\odot}$, the monoid homomorphism $\mathscr{H}(\odot, f_x, f_y)$ maps a pair of collections of type $\mathbb{T}_{\oplus}(\alpha)$ and $\mathbb{T}_{\otimes}(\beta)$ to a value of type $\mathbb{T}_{\odot}$. The additional algebraic law (3) implies that:

$$\mathscr{H}(\odot, f_x)X \odot \mathscr{H}(\odot, f_y)Y = \mathscr{H}(\odot, f_y)Y \odot \mathscr{H}(\odot, f_x)X \qquad \text{for all } X \text{ and } Y$$

which is required for asserting the associativity law (Eq. (2)).

By considering the different cases for X and Y, Eq. (4) is equivalent to the following equations for $H = \mathscr{H}(\odot, f_x, f_y)$:

$$\begin{aligned}
H(X \oplus X', Y \otimes Y') &= H(X,Y) \odot H(X',Y') \\
H(\mathbb{U}_{\oplus}(x), \mathbb{U}_{\otimes}(y)) &= f_x(x) \odot f_y(y) \\
H(\mathbb{U}_{\oplus}(x), \mathbf{1}_{\otimes}) &= f_x(x) \\
H(\mathbf{1}_{\oplus}, \mathbb{U}_{\otimes}(y)) &= f_y(y) \\
H(\mathbf{1}_{\oplus}, \mathbf{1}_{\otimes}) &= \mathbf{1}_{\odot}
\end{aligned}$$

provided that $\oplus \preceq \odot$, $\otimes \preceq \odot$, and Eq. (3) are satisfied. Section 4 defines the coGroup operation (a generalized join) as a binary homomorphism that satisfies Eq. (3).

4 The Monoid Algebra

The main goal of our work is to translate declarative data analysis queries to efficient programs that can run on various DISC platforms. Experience with the relational database technology has shown that this translation process can be made easier if we first translate the queries to an algebraic form that is equivalent to the query and then translate the algebraic form to a physical plan consisting of physical operations supported by the underlying DISC platform.

Our algebra consists of a very small set of operations that capture most features of many query languages and can be translated to efficient physical plans that can run on many DISC platforms. We intentionally use only one higher-order homomorphic operation in our algebra, namely cMap (flatten-map), which is a monoid homomorphism from a collection monoid to a possibly different collection monoid, to simplify normalization and optimization of algebraic terms. The cMap operation captures data parallelism, where each processing node evaluates the same code (the cMap functional argument) in parallel on its own data partition. The groupBy operation, on the other hand, re-shuffles the data across the processing nodes based on the group-by key, so that data with the same key are sent to the same processing node. The coGroup operation is a groupBy over two collections, so that data with the same key from both collections are sent to the same node to be joined. By moving all computations to cMap, our algebra detaches data distribution (specified by groupBy and coGroup) from data processing (cMap). This separation simplifies program optimization considerably. For example, as we will see in Section 9, query unnesting is done using just one rule, because there is only one place that a nested query can occur: at the cMap functional argument. In terms of divide-and-conquer computations, the groupBy and coGroup operations correspond to the divide part and a cMap to the conquer part.

Fig. 2 gives the syntax of the monoid algebra. A collection type in this algebra is associated with a collection monoid $\oplus$ and is represented as $\mathbb{T}_{\oplus}(\alpha)$. We use the shorthands $\{\!\{t\}\!\}$ for $\mathbb{T}_{\uplus}(t)$ and $[t]$ for $\mathbb{T}_{++}(t)$. The monoid algebraic operations are explained next.

The cMap operation. The first operation, cMap (better known as concat-map or flatten-map), generalizes the select, project, join, and unnest operations of the nested relational algebra. Given two collection monoids $\oplus$ and $\otimes$ with $\otimes \preceq \oplus$ and two arbitrary types α and β, the operation cMap(f, X) maps a collection X of type $\mathbb{T}_{\otimes}(\alpha)$ to a collection of

types: $t ::=$ basic $\mid \mathbb{T}_\oplus(t) \mid [t] \mid \{\!\{t\}\!\} \mid () \mid (t) \mid t_1 \times t_2 \mid t_1 + t_2$
variables: v
expressions:

$$\begin{array}{lll}
e \;::= & \text{cMap}(\lambda v.\, e_2,\, e_1) \mid \text{groupBy}(e) \mid \text{orderBy}(e) & \text{// operations on collections}\\
\mid & \text{coGroup}(e_1, e_2) \mid \text{reduce}(\oplus, e) \mid \text{repeat}(\lambda v.\, e_2,\, e_1, e_3) & \text{// operations on collections}\\
\mid & (e_1, e_2) \mid \pi_1(e) \mid \pi_2(e) & \text{// products}\\
\mid & \textbf{inL}(e) \mid \textbf{inR}(e) \mid \textbf{case}\; e\; \textbf{of}\; \textbf{inL}(v_1) \Rightarrow e_1 \mid \textbf{inR}(v_2) \Rightarrow e_2 & \text{// sums}\\
\mid & e_1 \oplus e_2 \mid \mathbf{1}_\oplus \mid \mathbb{U}_\oplus(e) & \text{// collection constructions}\\
\mid & \textbf{if}\; e_1\; \textbf{then}\; e_2\; \textbf{else}\; e_3 \mid () \mid (e) \mid v \mid \mathit{const} \mid \ldots & \text{// other operations}
\end{array}$$

Fig. 2. Syntax of the Monoid Algebra

type $\mathbb{T}_\oplus(\beta)$ by applying the function f of type $\alpha \to \mathbb{T}_\oplus(\beta)$ to each element of X, yielding one collection for each element, and then by merging these collections to form a single collection of type $\mathbb{T}_\oplus(\beta)$. It is expressed as follows as a monoid homomorphism:

$$\text{cMap}(f, X) \;\triangleq\; \mathscr{H}(\oplus, f)\, X \tag{5}$$

For $X = \mathbb{U}_\otimes(x_1) \otimes \cdots \otimes \mathbb{U}_\otimes(x_n)$, cMap$(f,X)$ computes $f(x_1) \oplus \cdots \oplus f(x_n)$, that is, it replaces $\mathbb{U}_\otimes$ with f and $\otimes$ with $\oplus$. For $\oplus = \otimes = \uplus$, cMap becomes a flatten-map over bags:

$$\begin{array}{rcl}
\text{cMap} & :: & (\alpha \to \{\!\{\beta\}\!\}) \to \{\!\{\alpha\}\!\} \to \{\!\{\beta\}\!\}\\
\text{cMap}(f, X \uplus Y) & = & \text{cMap}(f, X) \uplus \text{cMap}(f, Y)\\
\text{cMap}(f, \{\!\{a\}\!\}) & = & f(a)\\
\text{cMap}(f, \{\!\{\,\}\!\}) & = & \{\!\{\,\}\!\}
\end{array}$$

That is, this cMap maps a bag $\{\!\{x_1, \ldots, x_n\}\!\}$ to $f(x_1) \uplus \cdots \uplus f(x_n)$. Many common distributed queries can be written using cMaps over bags:

$$\begin{array}{rcl}
\text{map} & :: & (\alpha \to \beta) \to \{\!\{\alpha\}\!\} \to \{\!\{\beta\}\!\}\\
\text{map}(f, X) & = & \text{cMap}(\lambda x.\, \{\!\{f(x)\}\!\}, X)\\
\text{flatten} & :: & \{\!\{\{\!\{\alpha\}\!\}\}\!\} \to \{\!\{\alpha\}\!\}\\
\text{flatten}(X) & = & \text{cMap}(\lambda s.\, s, X)\\
\bowtie_p & :: & ((\alpha, \beta) \to \text{boolean}) \to \{\!\{\alpha\}\!\} \to \{\!\{\beta\}\!\} \to \{\!\{(\alpha, \beta)\}\!\}\\
X \bowtie_p Y & = & \text{cMap}(\lambda x.\, \text{cMap}(\lambda y.\, \textbf{if}\; p(x,y)\; \textbf{then}\; \{\!\{(x,y)\}\!\}\; \textbf{else}\; \{\!\{\,\}\!\},\; Y),\; X)
\end{array}$$

But cMap can also capture mappings between different collection types, such as from lists to bags. For example, a list can be converted to a bag using the following list homomorphism:

$$\begin{array}{rcl}
\text{list2bag} & :: & [\alpha] \to \{\!\{\beta\}\!\}\\
\text{list2bag}(X) & = & \text{cMap}(\lambda x.\, \{\!\{x\}\!\}, X)
\end{array}$$

Cascaded cMaps can be fused into a single nested cMap using the following law:

$$\text{cMap}(f, \text{cMap}(g, S)) \to \text{cMap}(\lambda x.\, \text{cMap}(f, g(x)), S) \tag{6}$$

This law is proven in Theorem A.1 in the Appendix.

The groupBy operation. Given an arbitrary type κ that supports value equality $(=)$, an arbitrary type α, and a collection X of type $\mathbb{T}_\oplus(\kappa \times \alpha)$, the operation groupBy$(X)$

groups the elements of X by their first component (the key) and returns a bag of type $\{\!\{\kappa \times \mathbb{T}_{\oplus}(\alpha)\}\!\}$. It can be expressed as follows as a monoid homomorphism:

$$\text{groupBy}(X) \triangleq \mathscr{H}(\Updownarrow_{\oplus}, \lambda(k,v).\,\{\!\{(k, \mathbb{U}_{\oplus}(v))\}\!\})\,X \tag{7}$$

The parametric monoid $\Updownarrow_{\oplus}$ merges groups associated with the same key using the monoid $\oplus$. It is expressed as follows using a set-former notation for bags:

$$\begin{aligned} X \Updownarrow_{\oplus} Y = & \{\,(k, a \oplus b) \,|\, (k,a) \in X,\, (k',b) \in Y,\, k = k'\,\} && \text{(equijoin between } X \text{ and } Y\text{)} \\ & \uplus\ \{\,(k,a) \,|\, (k,a) \in X,\, \forall (k',b) \in Y : k' \neq k\,\} && (\subseteq X \text{ not joined with } Y) \\ & \uplus\ \{\,(k,b) \,|\, (k,b) \in Y,\, \forall (k',a) \in X : k' \neq k\,\} && (\subseteq Y \text{ not joined with } X) \end{aligned} \tag{8}$$

That is, $X \Updownarrow_{\oplus} Y$ is a full outer join between X and Y. If the two arguments of $\Updownarrow_{\oplus}$ have distinct keys, then so does its result; and therefore, so does the result of groupBy. It is easy to see that, in the range of $\Updownarrow_{\oplus}$ (i.e, the set of key-value collections with distinct keys), the operation $\Updownarrow_{\oplus}$ is associative with identity $\{\!\{\,\}\!\}$. For a bag X of type $\{\!\{\kappa \times \alpha\}\!\}$, groupBy$(X)$ returns a bag of type $\{\!\{\kappa \times \{\!\{\alpha\}\!\}\}\!\}$. In that case, the groupBy definition is equivalent to the following equations:

$$\begin{aligned} \text{groupBy}(X \uplus Y) &= \text{groupBy}(X) \Updownarrow_{\uplus} \text{groupBy}(Y) \\ \text{groupBy}(\{\!\{(k,a)\}\!\}) &= \{\!\{(k, \{\!\{a\}\!\})\}\!\} \\ \text{groupBy}(\{\!\{\,\}\!\}) &= \{\!\{\,\}\!\} \end{aligned}$$

For example, groupBy$(\{\!\{(1,\text{“}a\text{”}), (2,\text{“}b\text{”}), (1,\text{“}c\text{”})\}\!\})$ returns $\{\!\{(1, \{\!\{\text{“}a\text{”},\text{“}c\text{”}\}\!\}), (2, \{\!\{\text{“}b\text{”}\}\!\})\}\!\}$. The groupBy result, which is of type $\{\!\{\kappa \times \{\!\{\alpha\}\!\}\}\!\}$, can be viewed as an indexed set (also known as a key-value map or a dictionary), which implements a partial function from κ to $\{\!\{\alpha\}\!\}$. In contrast to standard indexed sets, in which the second map overrides the first when two maps are combined, our combining operator $\Updownarrow_{\oplus}$ merges the ranges of overlapping maps using the monoid $\oplus$.

The monoid $\Updownarrow_{\oplus}$ is commutative and/or idempotent if $\oplus$ is commutative and/or idempotent. Note also that, for a bag X, groupBy(X) is different from nest(X), defined as:

$$\text{nest}(X) = \text{cMap}(\lambda(k,x).\,\{\!\{(k, \text{cMap}(\lambda(k',x').\, \textbf{if}\ k = k'\ \textbf{then}\ \{\!\{x'\}\!\}\ \textbf{else}\ \{\!\{\,\}\!\},\, X))\}\!\}, X)$$

Although both operations have the same type, the nest result may contain duplicate entries for a key k. Unlike nest, unnesting a groupBy over a bag X returns the input bag X:

$$\text{unnest}(\text{groupBy}(X)) = X \tag{9}$$

where unnest$(X) = \text{cMap}(\lambda(k,s).\,\text{cMap}(\lambda x.\,\{\!\{(k,x)\}\!\}, s), X)$. Eq. (9) is proven in Theorem A.2 in the Appendix. This law does not hold for lists, since groupBy converts a list to a bag. In addition, two cascaded groupBys can be fused to one groupBy using the law:

$$\text{groupBy}(\text{groupBy}(X)) = \text{cMap}(\lambda(k,s).\,\{\!\{(k, \{\!\{s\}\!\})\}\!\}, \text{groupBy}(X)) \tag{10}$$

This law is proven in Theorem A.3 in the Appendix.

Finally, the introduction of the groupBy operation justifies the reason for not using set collections and set homomorphisms; a bag X can be converted to a set (i.e., a bag with no duplicates) using a groupBy:

$$\text{distinct}(X) = \text{cMap}(\lambda(k,s).\{\!\{k\}\!\}, \text{groupBy}(\text{cMap}(\lambda x.\,\{\!\{(x,x)\}\!\}, X))) \tag{11}$$

The orderBy operation. If we order a bag of type $\{\!\!\{\kappa \times \alpha\}\!\!\}$ by its key κ (which must support a total order $\leq$), we should get a list of type $[\kappa \times \{\!\!\{\alpha\}\!\!\}]$, rather than $[\kappa \times \alpha]$, because, in general, there may be multiple values of type α associated with the same key, and these values have to be put into a bag for this operation to be unambiguous. Hence, we have chosen to define the orderBy operation in a way similar to groupBy:

$$\text{orderBy}(X) \triangleq \mathscr{H}(\Uparrow_{\oplus}, \lambda(k,v).\,[(k, \mathbb{U}_{\oplus}(v))])\,X \tag{12}$$

where $\oplus$ is a monoid. The monoid $\Uparrow_{\oplus}$ merges two sorted sequences of type $[\kappa \times \mathbb{T}_{\oplus}(\alpha)]$ to create a new sorted sequence. It can be expressed as follows using a set-former notation for lists (equivalent to list comprehensions):

$$(X_1 \mathbin{+\!\!+} X_2) \Uparrow_{\oplus} Y \;=\; X_1 \Uparrow_{\oplus} (X_2 \Uparrow_{\oplus} Y) \tag{13a}$$

$$\begin{aligned}[(k,v)] \Uparrow_{\oplus} Y \;=\; & [\,(k',w)\,|\,(k',w) \in Y, k' < k\,] \\ & \mathbin{+\!\!+} [(k, v \oplus (\oplus/[\,w\,|\,(k',w) \in Y, k' = k\,])\,)] \\ & \mathbin{+\!\!+} [\,(k',w)\,|\,(k',w) \in Y, k' > k\,]\end{aligned} \tag{13b}$$

$$[\,] \Uparrow_{\oplus} Y \;=\; Y \tag{13c}$$

where $\oplus/s$ reduces the lists s of type $[\mathbb{T}_{\oplus}(\alpha)]$ to a value of type $\mathbb{T}_{\oplus}(\alpha)$ using $\oplus$. That is, Eq. (13b) inserts the pair (k,v) into the sorted list Y, deriving a sorted list. Theorem A.4 in the Appendix shows that, in the range of $\Uparrow_{\oplus}$ (the set of sorted lists of pairs), $\Uparrow_{\oplus}$ is a monoid with identity $[\,]$. In addition, even though $\mathbin{+\!\!+} \prec \uplus$, we have $\uplus \preceq \Uparrow_{\uplus}$ since $\Uparrow_{\uplus}$ is commutative, which makes orderBy a homomorphism for both lists and bags. Finally, although a list of type $[\kappa \times \alpha]$ is sorted to a list of type $[\kappa \times [\alpha]]$, it can always be flattened to a $[\kappa \times \alpha]$.

The reduce operation. Aggregations are captured by the operation $\text{reduce}(\oplus, X)$, where $\otimes \preceq \oplus$, which aggregates a collection X of type $\mathbb{T}_{\otimes}(t)$ using the non-collection monoid $\oplus$ of type t:

$$\text{reduce}(\oplus, X) \triangleq \mathscr{H}(\oplus, \lambda x.x)\,X \tag{14}$$

For example, $\text{reduce}(+, \{\!\!\{1,2,3\}\!\!\}) = 6$.

The coGroup operation. Although a join can be expressed as a nested cMap, we provide a special homomorphic operation for equi-joins and outer joins. The operation $\text{coGroup}(X,Y)$ between a collection X of type $\mathbb{T}_{\oplus}(\kappa \times \alpha)$ and a collection Y of type $\mathbb{T}_{\otimes}(\kappa \times \beta)$ over their first component of type κ (the join key) returns a collection of type $\{\!\!\{\kappa \times (\mathbb{T}_{\oplus}(\alpha) \times \mathbb{T}_{\otimes}(\beta))\}\!\!\}$. It can be expressed as a binary homomorphism as follows:

$$\begin{aligned}\text{coGroup}(X,Y) \triangleq \mathscr{H}(& \Updownarrow_{\oplus \star \otimes}, \\ & \lambda(k,x).\,\{\!\!\{(k, (\mathbb{U}_{\oplus}(x), \mathbf{1}_{\otimes}))\}\!\!\}, \\ & \lambda(k,y).\,\{\!\!\{(k, (\mathbf{1}_{\oplus}, \mathbb{U}_{\otimes}(y)))\}\!\!\})\;(X,Y)\end{aligned} \tag{15}$$

where $\Updownarrow$ is the same monoid we used for groupBy, but is now parameterized by the product of two collections monoids, $\oplus \star \otimes$. The product of two monoids $\oplus \star \otimes$ is a monoid with identity $(\mathbf{1}_{\oplus}, \mathbf{1}_{\otimes})$ and a binary operation $\oplus \star \otimes$, such that $(x_1,y_1)(\oplus \star \otimes)(x_2,y_2) = (x_1 \oplus x_2, y_1 \otimes y_2)$. For bags, the coGroup operation has type $\{\!\!\{\kappa \times \alpha\}\!\!\} \times \{\!\!\{\kappa \times \beta\}\!\!\} \rightarrow$

$\{\!\{\kappa \times (\{\!\{\alpha\}\!\} \times \{\!\{\beta\}\!\})\}\!\}$ and is equivalent to:

$$\begin{array}{rcll}
\text{coGroup}(X_1 \uplus X_2, Y_1 \uplus Y_2) & = & \text{coGroup}(X_1,Y_1) \Updownarrow_{\uplus\star\uplus} \text{coGroup}(X_2,Y_2) & \\
\text{coGroup}(\{\!\{(k,a)\}\!\}, \{\!\{(k',b)\}\!\}) & = & \{\!\{(k,(\{\!\{a\}\!\},\{\!\{b\}\!\}))\}\!\} & \text{if } k = k' \\
\text{coGroup}(\{\!\{(k,a)\}\!\}, \{\!\{(k',b)\}\!\}) & = & \{\!\{(k,(\{\!\{a\}\!\},\{\!\{\,\}\!\})),\, (k',(\{\!\{\,\}\!\},\{\!\{b\}\!\}))\}\!\} & \text{if } k \neq k' \\
\text{coGroup}(\{\!\{(k,a)\}\!\}, \{\!\{\,\}\!\}) & = & \{\!\{(k,(\{\!\{a\}\!\},\{\!\{\,\}\!\}))\}\!\} & \\
\text{coGroup}(\{\!\{\,\}\!\}, \{\!\{(k,b)\}\!\}) & = & \{\!\{(k,(\{\!\{\,\}\!\},\{\!\{b\}\!\}))\}\!\} & \\
\text{coGroup}(\{\!\{\,\}\!\}, \{\!\{\,\}\!\}) & = & \{\!\{\,\}\!\} &
\end{array}$$

For example, $\text{coGroup}(\{\!\{(1,\text{“}a\text{”}),(2,\text{“}b\text{”}),(1,\text{“}c\text{”})\}\!\}, \{\!\{(1,\text{“}d\text{”}),(2,\text{“}e\text{”}),(3,\text{“}f\text{”})\}\!\})$ returns $\{\!\{(1,(\{\!\{\text{“}a\text{”},\text{“}c\text{”}\}\!\},\{\!\{\text{“}d\text{”}\}\!\})),(2,(\{\!\{\text{“}b\text{”}\}\!\},\{\!\{\text{“}e\text{”}\}\!\})),(3,(\{\!\{\,\}\!\},\{\!\{\text{“}f\text{”}\}\!\}))\}\!\}$.

From Eq. (2), for $X = X \oplus \mathbf{1}_\oplus$ and $Y = \mathbf{1}_\otimes \otimes Y$, we have:

$$\text{coGroup}(X,Y) \;=\; \text{coGroup}(X,\mathbf{1}_\otimes) \Updownarrow_{\oplus\star\otimes} \text{coGroup}(\mathbf{1}_\oplus,Y) \tag{16}$$

Finally, it can be proven by induction that:

$$\text{coGroup}(X,\mathbf{1}_\otimes) \;=\; \text{cMap}(\lambda(k,s).\,\{\!\{(k,(s,\mathbf{1}_\otimes))\}\!\},\, \text{groupBy}(X)) \tag{17a}$$

$$\text{coGroup}(\mathbf{1}_\oplus,Y) \;=\; \text{cMap}(\lambda(k,s).\,\{\!\{(k,(\mathbf{1}_\oplus,s))\}\!\},\, \text{groupBy}(Y)) \tag{17b}$$

The repeat operation. This is the only non-homomorphic operation in the monoid algebra. It is a fix-point operation that is used to capture data analysis algorithms that require iteration, such as data clustering and PageRank. Given a collection X of type $\mathbb{T}_\oplus(\alpha)$, a function f of type $\mathbb{T}_\oplus(\alpha) \rightarrow \mathbb{T}_\oplus(\alpha \times \text{boolean})$, and an integer n, $\text{repeat}(f,X,n)$ returns a collection of type $\mathbb{T}_\oplus(\alpha)$ and is defined as follows:

$$\begin{array}{rcl}
\text{repeat}(f,X,n) & \triangleq & \textbf{let}\; s = f(X) \\
 & & \textbf{in}\;\; \textbf{if}\; n \leq 0 \,\vee\, \neg\text{reduce}(\vee,\Pi_2(s)) \\
 & & \quad\;\; \textbf{then}\; \Pi_1(s) \\
 & & \quad\;\; \textbf{else}\; \text{repeat}(f,\Pi_1(s),n-1)
\end{array}$$

where $\Pi_1(s) = \text{cMap}(\lambda(x,b).\,\mathbb{U}_\oplus(x),\, s)$ returns a collection of type $\mathbb{T}_\oplus(\alpha)$ that contains the first elements of s, and, similarly, $\Pi_2(s)$ returns the second elements of s. The repetition stops if the number of remaining repetitions is zero or when all the boolean values returned by s are false.

4.1 Type Rules

Fig. 3 gives some of the type rules for the monoid algebra. Each type rule has two parts separated by a horizontal line: the part above the line contains the premises and the part below the line is the conclusion. We use the judgment $\Gamma \vdash e : t$ to indicate that e has type t under the environment Γ, which binds variables to types. The notation $v : t \in \Gamma$ checks if there is a binding from v to t in Γ, while $\Gamma, v : t$ extends the environment Γ with a new binding from v to t.

In abstract algebra, the composition of monoid homomorphisms is also a monoid homomorphism. The type rules of the monoid algebra though, shown in Fig. 3, allow terms that are not always monoid homomorphisms. More specifically, the type of the groupBy operation in Eq. (18g) is a bag (a $\mathbb{T}_\uplus$), instead of a $\mathbb{T}_{\Updownarrow_\oplus}$. Similarly, the type of the coGroup op-

$$\frac{v : t \in \Gamma}{\Gamma \vdash v : t} \tag{18a}$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \tag{18b}$$

$$\frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_i(e) : t_i} \tag{18c}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{T}_{\oplus}(t) \quad \Gamma \vdash e_2 : \mathbb{T}_{\oplus}(t)}{\Gamma \vdash e_1 \oplus e_2 : \mathbb{T}_{\oplus}(t)} \tag{18d}$$

$$\frac{\Gamma \vdash e : \mathbb{T}_{\otimes}(t) \quad t = \mathbb{T}_{\oplus} \quad \otimes \preceq \oplus}{\Gamma \vdash \text{reduce}(\oplus, e) : t} \tag{18e}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{T}_{\otimes}(t_1) \quad \Gamma, v : t_1 \vdash e_2 : \mathbb{T}_{\oplus}(t_2) \quad \otimes \preceq \oplus}{\Gamma \vdash \text{cMap}(\lambda v.\, e_2,\, e_1) : \mathbb{T}_{\oplus}(t_2)} \tag{18f}$$

$$\frac{\Gamma \vdash e : \mathbb{T}_{\oplus}(t_1 \times t_2)}{\Gamma \vdash \text{groupBy}(e) : \{\!\{ t_1 \times \mathbb{T}_{\oplus}(t_2) \}\!\}} \tag{18g}$$

$$\frac{\Gamma \vdash e : \mathbb{T}_{\oplus}(t_1 \times t_2)}{\Gamma \vdash \text{orderBy}(e) : [t_1 \times \mathbb{T}_{\oplus}(t_2)]} \tag{18h}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{T}_{\oplus}(t \times t_1) \quad \Gamma \vdash e_2 : \mathbb{T}_{\otimes}(t \times t_2)}{\Gamma \vdash \text{coGroup}(e_1, e_2) : \{\!\{ t \times (\mathbb{T}_{\oplus}(t_1) \times \mathbb{T}_{\otimes}(t_2)) \}\!\}} \tag{18i}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_3 : \text{int} \quad \Gamma \vdash e_1 : \mathbb{T}_{\oplus}(t) \\ \Gamma, v/\mathbb{T}_{\oplus}(t) \vdash e_2 : \mathbb{T}_{\oplus}(t \times \text{boolean})\end{array}}{\Gamma \vdash \text{repeat}(\lambda v.\, e_2, e_1, e_3) : \mathbb{T}_{\oplus}(t)} \tag{18j}$$

Fig. 3. Some Type Rules for the Monoid Algebra

eration in Eq. (18i) is a bag, instead of a $\mathbb{T}_{\Updownarrow_{\oplus \star \otimes}}$. That is, given that $\uplus \prec \Updownarrow_{\uplus}$, we have down-coerced (in terms of the monoid order) the results of groupBy and coGroup by erasing their distinct-key laws. This is equivalent to coercing a set to a bag. These coercions were necessary to allow us to capture non-homomorphic queries. Without this coercion, most terms of the form $\text{cMap}(f, \text{groupBy}(X))$ would not have been type-correct, indicating that cMap does not always distribute over $\Updownarrow_{\uplus}$. For example, the term $\text{cMap}(\lambda\,(k,x).\, \{\!\{x\}\!\}, \text{groupBy}(X))$ would not have been type-correct because this cMap would be a homomorphism from $\Updownarrow_{\uplus}$ to $\uplus$, which is invalid because $\Updownarrow_{\uplus} \not\preceq \uplus$. By down-coercing the groupBy result, terms of the form $\text{cMap}(f, \text{groupBy}(X))$ are accepted as type-correct but may not be homomorphisms.

5 The Monoid Algebra as a Formal Basis for Data-Centric Distributed Computing

Data parallelism is a form of parallel processing in which the same task operates on different data in parallel. By processing data in parallel, we can achieve a parallel speedup. The result of a data-parallel computation though must be independent of the way we divide the data into partitions and the way we combine the partial results of processing these partitions to obtain the final result. These requirements suggest that data-parallel computations should be associative so that they can apply to data partitions in parallel and can successively combine the intermediate results of computations in arbitrary ways. Data placement is an essential part of a data-parallel algorithm, since the mapping of data to processors determines the locality of data access, which is the dominant factor for the performance of data-parallel programs. DISC frameworks are essentially data-parallel frameworks on clusters of shared-nothing computers connected through a high-speed network. These clusters are typically built on top of cheap commodity hardware, with inconsistent performance across processing nodes, for which failure is not uncommon. Due to the volume of data, complex data analysis programs can involve thousands of nodes and can take hours to complete. Since the probability of a failure occurring during a long-running data analysis task is relatively high, DISC frameworks support fault tolerance so that a task does not have to restart if one of the processing nodes fails. DISC frameworks achieve data locality by distributing the data across the processing nodes through a distributed file system (DFS) and by sending the computation to the processing nodes that hold a copy of the data locally. Finally, in contrast to traditional database query processing, DISC systems organize computations for sequential reads (full scans), not random access, to take advantage of the high data throughput of these scans in a DFS.

In this section, we give evidence to support our claim that the monoid algebra is a good basis for DISC computations. Mapping monoid algebra terms to distributed operations supported by a DISC platform is a very complex process that needs to consider alternative algorithms to implement each algebraic operation, to take into account the available hardware resources, to adjust various configuration parameters for good performance, etc. In this section though, to support our claim on the appropriateness of the monoid algebra for DISC computations, we provide a naive data-parallel execution model for the monoid algebra that has a decent performance on a typical DISC platform. Although the cost of execution may depend on many factors, we simplify our analysis by considering only the amount of data shuffled among compute nodes across the network, since this is the dominant cost factor for data-centric distributed processing. Furthermore, we assume that, in case of nested collections, only the operations on the outer collection may cause data shuffling because each inner collection is stored entirely in a single processing node. Using Spark's terminology, we classify the monoid algebraic operations on outer collections into two categories: those that cause data shuffling (called transformations with wide dependencies in Spark), namely groupBy, orderBy, coGroup, and reduce, and those that do not, namely cMap. We define a stage to be a sub-series of operations that contains exactly one operation that requires data shuffling. In other words, the number of stages in a term is equal to the number of transformations with wide dependencies. The repeat algebraic operation is a fix-point operation that repeats the evaluation of its functional argument, and, thus, may consists of multiple stages.

For example, a Map-Reduce job (Dean and Ghemawat, 2004), which consists of a map function m of type $k_1 \times \alpha \to \{\!\{k_2 \times \beta\}\!\}$ and a reduce function r of type $k_2 \times \{\!\{\beta\}\!\} \to \{\!\{k_3 \times \gamma\}\!\}$, is expressed in the monoid algebra as follows:

$$\text{mapReduce}(m,r)(X) \quad = \quad \text{cMap}(r, \text{groupBy}(\text{cMap}(m,X)))$$

That is, this term requires one stage only because it uses one operation that causes data shuffling (groupBy).

To show the effectiveness of the monoid algebra for expressing DISC computations, we compare its naive evaluation with a typical implementation of the Bulk Synchronous Parallelism (BSP) programming model (Valiant, 1990). A BSP computation consists of a sequence of supersteps. Each superstep is evaluated in parallel by every peer participating in the BSP computation. A superstep consists of three phases: a local computation, a process communication, and a barrier synchronization. During the local computation phase of a superstep, each peer has access to the messages sent by other peers during the previous superstep. It can also send messages to other peers during the process communication phase, to be read at the next superstep. The barrier synchronization phase synchronizes all peers to make sure that they have received all the sent messages before the next superstep. The cost of a BSP computation depends on the cost of the longest running local computation, the cost of global communication between the processors, and the cost of the barrier synchronization. We are interested in a data-centric BSP framework for which data shuffling caused by the global communication is the dominant cost factor. Therefore, our comparison is based on two metrics: the number of supersteps and the total amount of data shuffled during all supersteps.

In an earlier work (Fegaras, 2012), we have shown that any term in the monoid algebra can be evaluated in BSP mode (more specifically, in Apache Hama (Apache Hama, 2017)) using just one superstep per stage. We now show the reverse: we translate any BSP program to a monoid algebra term in such a way that the number of stages required to execute this term is equal to the number of supersteps needed to execute the BSP program, and, more importantly, both programs shuffle the same amount of data across the network. BSP is a general model for synchronous parallel processing. Most DISC frameworks, including Map-Reduce, Spark, and Flink, require similar peer synchronization between stages. Asynchronous HPC frameworks, such as MPI, on the other hand, are not typically used as a basis for DISC systems because it is very hard to achieve fault tolerance on such frameworks. Furthermore, asynchronous programs may cause deadlocks and livelocks, while synchronization barriers cannot create circular data dependencies. By showing that our monoid algebra can capture data-centric BSP computations efficiently, we give a strong evidence that it can also capture most of the current DISC systems efficiently.

There are many different ways of implementing the BSP model, since this model does not fully specify the computation phase. In a functional setting, we can specify the computation as a pure function to be executed locally by each BSP peer. This compute function may be of the type $id \times \{\!\{m\}\!\} \times \sigma \rightarrow \{\!\{id \times m\}\!\} \times \sigma \times \text{boolean}$, where id is the type of the peer id, m is the message type, and σ is the state type. Each peer retains a local state of type σ. At each superstep, each peer, with id id and local state $state$, receives a bag of incoming messages ms and calls compute$(id, ms, state)$, which returns a bag of outgoing messages of type $\{\!\{id \times m\}\!\}$, a new state, and a boolean value. The compute function is evaluated repeatedly by each peer until the returned boolean value is true for all peers. Using pseudo-code, the BSP process is as follows:

```
for each peer i do:              // do in parallel at each peer
    msgs ← { }
    state[i] ← initial_state[i]
    repeat
        (new_msgs, state[i], exit[i]) ← compute(i,msgs,state[i])
        send new_msgs to peers
        wait for barrier synchronization
        msgs ← receive messages from peers
    until for all peers j: exit[j]
```

For a data-centric BSP computation, the initial_state[i] of a peer i may contain partitions of the input datasets stored locally at the peer i. Note that a peer is a virtual computational unit; multiple peers may be assigned to a single computation node, or the code and data of a single peer may be distributed across multiple computation nodes. We now simulate this program using the monoid algebra:

$$\begin{array}{l}
\mathrm{repeat}(\lambda P.\\
\qquad \mathrm{cMap}(F, \mathrm{groupBy}(\mathrm{cMap}(\lambda(id,ms,s).\,\mathbf{let}\ (ps,s',exit) = \mathrm{compute}(id,ms,s)\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ \{\!\{(id, \mathbf{inL}((s',\ \mathbf{not}\ exit)))\}\!\}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \uplus\ \mathrm{cMap}(\lambda(i,m).\,\{\!\{(i,\mathbf{inR}(m))\}\!\}, ps),\\
\qquad\qquad\qquad\qquad\qquad\quad P))),\\
\qquad \mathrm{cMap}(\lambda(id,s).\,\{\!\{(id,\{\!\{\ \}\!\},s)\}\!\}, Peers))
\end{array}$$

The repetition will stop when the exit value returned by compute is true for all peers (since the max number of repetitions is unbounded). The initial repetition dataset is *Peers*, which has type $\{\!\{id \times \sigma\}\!\}$ and associates an initial local state to each peer. Since each initial state is local to each peer in the original BSP program, we assume that the dataset *Peers* is already partitioned and distributed across the peers. Function F is from $id \times \{\!\{(\sigma \times \text{boolean}) + m\}\!\}$ to $\{\!\{(id \times \{\!\{m\}\!\} \times \sigma) \times \text{boolean}\}\!\}$, with $F(id, vs)$ equal to:

$$\begin{array}{l}
\mathrm{cMap}(\lambda(s,exit).\,\{\!\{((id, \mathrm{cMap}(\lambda v.\,\mathbf{case}\ v\,\mathbf{of}\ \mathbf{inL}(w) \Rightarrow \{\!\{\ \}\!\} \mid \mathbf{inR}(m) \Rightarrow \{\!\{m\}\!\}, vs), s),\ exit)\}\!\}\\
\qquad \mathrm{cMap}(\lambda v.\,\mathbf{case}\ v\,\mathbf{of}\ \mathbf{inL}(w) \Rightarrow \{\!\{w\}\!\} \mid \mathbf{inR}(m) \Rightarrow \{\!\{\ \}\!\}, vs)
\end{array}$$

We can see that the evaluation of each repetition step requires one stage only because there is only one groupBy operation in the repeat step. Therefore, the number of stages is equal to the number of supersteps. Furthermore, the data processed by groupBy consist of the new local states and the outgoing messages. But each new state is sent to the same node that generated the state, since it is grouped by the same id. This means that the only data shuffled across the network are the outgoing messages to other peers, which are equal to the data shuffled by the original BSP program.

It is now easy to show that our monoid algebra can also capture vertex-centric graph-parallel programs efficiently. Vertex-centric graph-parallel programming is a new popular framework for large-scale graph processing, introduced by Google's Pregel (Malewicz *et al.*, 2010) and now available as an open-source project by Apache Giraph (Apache Giraph, 2017). A common characteristic of these frameworks is that they are all based on the BSP model. In a vertex-centric specification, a graph is a bag of vertices, $\{\!\{N\}\!\}$, where the type N of a vertex can be specified as $id \times \alpha \times \{\!\{id \times \beta\}\!\}$. That is, a vertex has a unique id, a state of type α, and a bag of outgoing edges, where each edge is associated with a destination vertex id and a state of type β. Then, a vertex-centric program is a function from $\{\!\{N\}\!\}$ to $\{\!\{N\}\!\}$, which, not only can change the vertex and edge states of the graph, but can also change the graph topology by removing existing edges and adding new edges. This specification can be easily captured by a BSP program for which an id corresponds to a vertex, rather than a peer. In this case, the BSP state type σ associated with a vertex is equal to the type N. Based on our simulation of general BSP programs, graph-parallel programs too can be implemented efficiently using our monoid algebra.

6 Comparison of the Monoid Algebra with Algebras based on Monads

Before we discuss how the monoid algebra compares with algebras based on monads, let's discuss what features a practical data-centric query language should support. Although the relational data model is based on sets of records, practical relational query languages are based on multisets (bags). SQL, for example, has mostly bag semantics, with the exception of a few operations that remove duplicates, such as, the ‘select distinct’ and ‘union distinct’ syntax, which have set semantics, and the ‘order by’ clause, which creates a sorted bag (a list). The reason that practical database query languages are mostly based on bag semantics is that removing duplicates after every operation is very expensive and often unnecessary. In addition, some operations may depend on the multiplicity of values, such as counting occurrences of values in a collection. Another problem with sets is that set operations defined using structural recursion on the union representation of sets must be idempotent. That is, non-idempotent operations, such as most aggregations, cannot be expressed as pure divide-and-conquer computations on sets, as we can see from the definition $\mathrm{sum}(X \cup Y) = \mathrm{sum}(X - Y) + \mathrm{sum}(Y)$. Some formal models and languages, such as FAD (Bancilhon *et al.*, 1987), have addressed this problem by using structural recursion on the disjoint union ($\sqcup$), defined only for disjoint sets, so that $\mathrm{sum}(X \sqcup Y) = \mathrm{sum}(X) + \mathrm{sum}(Y)$, provided that $X \cap Y = \emptyset$. But such operations are not suitable for data parallelism since they are not divide-and-conquer computations.

DISC frameworks too are mostly based on bag semantics, but they also support a small number of sorting and duplicate-removal operations. In Map-Reduce, for example, the map phase is over a distributed bag of key-values pairs, while the reduce phase is over a distributed indexed set, which, in our framework, is represented as the carrier of the monoid $\Updownarrow_{\uplus}$. In Spark, the only collection type is the RDD (Resilient Distributed Dataset), which is a bag of values, but many Spark methods are from the RDD subclass PairRDDFunctions, which represents a bag of key-value pairs. Spark supports operations, such as groupByKey, reduceByKey, sortBy, and cogroup, that can be directly translated to the monoid algebra. Like most practical database query languages and DISC frameworks, the monoid algebra does not support sets; instead, one may use the groupBy operation to remove duplicates from a bag.

The rest of this section compares the monoid algebra with algebras based on monads and shows that monad algebras have a number of shortcomings as a formal basis for practical query languages for DISC systems. Most notably, monad algebras require extensions to capture coercions between collections, aggregations, sorting, grouping, and outer-joins. The monoid algebra does not have these shortcomings. In general, adding extensions to fill up the missing functionality requires the introduction of extra laws, which may complicate the basic formal framework and make optimization harder.

Collection monads (also called ringads) are monads (Wadler, 1990) extended with an empty value and a merge function that form a monoid. Thus, a collection monad must provide the following functions:

$$\begin{array}{rcl@{\qquad\qquad}rcl}
\mathrm{map} & : & (\alpha \rightarrow \beta) \rightarrow M(\alpha) \rightarrow M(\beta) & \mathrm{empty} & : & M(\alpha) \\
\mathrm{unit} & : & \alpha \rightarrow M(\alpha) & \mathrm{merge} & : & M(\alpha) \rightarrow M(\alpha) \rightarrow M(\alpha) \\
\mathrm{concat} & : & M(M(\alpha)) \rightarrow M(\alpha) & & &
\end{array}$$

The monad extension operator ext of type $(\alpha \rightarrow M(\beta)) \rightarrow M(\alpha) \rightarrow M(\beta)$, which is equivalent to the bind operation $\gg\!=$ of type $M(\alpha) \rightarrow (\alpha \rightarrow M(\beta)) \rightarrow M(\beta)$, is derived from the other monad operations as $\mathrm{ext}(f) = \mathrm{concat} \circ \mathrm{map}(f)$. That is, the type of ext is more restrictive than the type of cMap of the monoid algebra, which is $(\alpha \rightarrow M(\beta)) \rightarrow N(\alpha) \rightarrow M(\beta)$, provided that the monoid of the collection type M obeys all the laws of the monoid of the collection type N. Hence, cMap is more expressive than ext because every ext operation can be expressed as a cMap using $\text{ext}(f)\,X = \text{cMap}(f,X)$, while a cMap from a collection N to a different collection M cannot be expressed as an ext.

Monad comprehensions (Wadler, 1990) is a declarative syntax for expressing monad operations that generalizes list comprehensions. Monad comprehensions can be easily translated to monad operations using rules similar to those for translating list comprehensions (Wadler, 1987). One important extension towards bringing comprehensions closer to a practical query language was the work on list comprehensions with group-by and order-by (Wadler and Peyton Jones, 2007). This work was generalized by (Giorgidze *et al.*, 2011) to support order-by and group-by syntax in any monad comprehension. Our monoid comprehensions support order-by and group-by syntax over heterogeneous collections without requiring any extension to the basic monoid homomorphism framework.

One difficulty in using collection monads and monad comprehensions as a practical query language is that, without extending them, we cannot mix different collection monads in the same monad comprehension, for example to join a bag with a list or to sort a bag into a list. This problem has been addressed by expressing conversion between lists, bags, and sets, as monad morphisms, thus supporting heterogeneous monad comprehensions (Wong, 2000). Based on the analysis in (Gibbons, 2016), these coercions between monad collections are monad morphisms if they are from poorer to richer members of the Boom hierarchy of types. Furthermore, these coercions must be monoid homomorphisms and must obey the monad morphism laws. Other coercions, such as from a bag to a list, are not ‘obvious’ and should be defined explicitly (Gibbons, 2016). Another extension that is required is the support for aggregations, since all practical query languages support total aggregation and group-by with aggregation. Aggregations on a collection monad can be defined as algebras for that monad. These aggregations, in addition to obeying the monad algebra laws, must be monoid homomorphisms (Gibbons, 2016).

The most important shortcoming of algebras based on monads is that they cannot capture grouping and sorting on bags without extending the algebra with additional operators that must obey certain laws. Although it is possible to express the group-by operation on a set X as a monad comprehension on sets (the subscript *Set* identifies the comprehension type):

$$\{\, (k, \{\, a \mid (k',a) \in X,\, k' = k \,\}_{Set}\,) \mid (k, \_) \in X \,\}_{Set}$$

a group-by on a bag X would require a bag2set coercion to remove duplicate keys:

$$\{\, (k, \{\, a \mid (k',a) \in X,\, k' = k \,\}_{Bag}\,) \mid k \in \text{bag2set}(\{\, k \mid (k, \_) \in X \,\}_{Bag}) \,\}_{Set}$$

Since bag2set creates a set, the outer monad comprehension must now be a set comprehension. But deriving a set comprehension for group-by instead of a bag comprehension is problematic because it prevents us from applying non-idempotent operations, such as count, to the result. Furthermore, we cannot down-coerce the resulting set to a bag without introducing an additional non-obvious operation. Hence, the group-by operation would require a special extension to monads. For example, the SQL-like comprehensions in (Giorgidze *et al.*, 2011) require various library additions to support order-by and group-by syntax in any monad comprehension. For group-by, the required addition is a new operation mgroupWith of type $(\alpha \to \tau) \to m\,\alpha \to m(m\,\alpha)$, which takes an explicit group-by key function of type $\alpha \to \tau$.

patterns: $p ::= v \mid (p_1, \ldots, p_n) \mid \text{inv}(p)$
qualifiers: $q ::= p \in e \mid \textbf{let}\, p = e \mid e \mid \textbf{group by}\, p\,[: e] \mid \textbf{order by}\, p\,[: e]$
expressions: $e ::= \ldots$ // monoid algebra expressions (in Fig. 2)
$\mid \{\, e \mid q_1, \ldots, q_n \,\}$ // monoid comprehensions
$\mid \oplus/e$ // reductions

Fig. 4. Syntax of Monoid Comprehensions

The monoid algebra and monoid comprehensions do not require any extensions in the form of added non-homomorphic operations to capture practical data-centric query languages (except for the fix-point operation 'repeat', which is also required for frameworks based on monads, if they need to capture iteration). Monoid comprehensions can work on heterogeneous collections, support group-by and order-by syntax in comprehensions for all collection types, and capture aggregations, without any extension to the formal framework on collection monoids and monoid homomorphisms. Monads on the other hand require numerous extensions to support the same features (monoidal structure, coercion functions as monad morphisms, and monoid homomorphisms for aggregations). Given the necessity of introducing these extensions to monads to capture practical data-centric query languages, as well as all the additional laws that these added operations must obey, it makes the choice of monoids as a formal basis for data-centric query languages better than monads. Furthermore, rather than extending monads with monoid homomorphisms it seems simpler to use monoid homomorphisms for all operations.

7 Monoid Comprehensions

Our primary query language for distributed data analysis is MRQL, to be described in Section 8, which is, as we will see, equivalent to monoid comprehensions. Since MRQL has a lot of syntactic sugar, we have decided to present our type and translation rules based on monoid comprehensions first, and then translate MRQL to monoid comprehensions.

The syntax and semantics of our monoid comprehensions have been influenced by previous work on list comprehensions with order-by and group-by (Wadler and Peyton Jones, 2007). Monoid comprehensions were first introduced in (Fegaras and Maier, 1995), but they are extended here to include well-behaved group-by and order-by qualifiers. Monoid comprehensions can work on heterogeneous collection types and are extended with group-by and order-by syntax that works on heterogeneous comprehensions. They do not support some extensions used in monad comprehensions, such as parallel (zip) qualifiers, because they are not suitable for bags, parenthesized qualifiers, because they are equivalent to nested comprehensions, refutable patterns, and user-defined group-by and order-by functions (Wadler and Peyton Jones, 2007).

Before we describe monoid comprehensions formally, let's consider an example of a monoid comprehension with group-by syntax. If we represent a sparse matrix M as a bag of triples, (v,i,j), for $v = M_{ij}$, then the matrix multiplication between the two sparse matrices X and Y can be expressed as follows:

$$\{\, (+/z, i, j) \mid (x,i,k) \in X,\ (y,k',j) \in Y,\ k = k',\ \textbf{let}\ z = x * y,\ \textbf{group by}\ (i,j) \,\}$$

which retrieves the values $X_{ik} \in X$ and $Y_{kj} \in Y$ for all i, j, k, and it sets $z = X_{ik} * Y_{kj}$. The group-by operation lifts each pattern variable defined before the group-by (except the group-by keys) from some type t to $\{\!\{t\}\!\}$, indicating that each such variable must now contain all the values associated with the same group-by key value. Consequently, after we group the values by the indexes i and j, the variable z is lifted to a bag of numerical values $X_{ik} * Y_{kj}$, for all k. Hence, $+/z$, which is a shorthand for reduce$(+, e)$, will sum up all these values, deriving $\sum_k X_{ik} * Y_{kj}$ for the ij element of the resulting matrix.

Fig. 4 describes the monoid comprehension syntax. The $\oplus/e$ syntax is a shorthand for reduce$(\oplus, e)$. The domain e of a generator qualifier $p \in e$ must be of a collection type $\mathbb{T}_\oplus(t)$, associated with a collection monoid $\oplus$. The generator draws elements from this collection and, each time, it binds the irrefutable pattern p to an element. The generator domains in a monoid comprehension can be of different collection types. The result of a monoid comprehension is a collection whose collection monoid is the maximum collection monoid (in terms of the monoid order) of all collection monoids used in the comprehension generators. A let-binding **let** $p = e$ binds the pattern p to the result of e. A qualifier e, called a filter, is a predicate of type boolean.

The group-by and order-by qualifiers use a pattern p and an optional expression e. If e is missing, it is taken to be p. The group-by operation groups all the pattern variables in the same comprehension that are defined before the group-by (except the variables in p) by the value of e (the group-by key), so that all variable bindings that result to the same key value are grouped together. After the group-by, p is bound to a group-by key and each one of these pattern variables is lifted to a collection of values. Note that patterns in a comprehension with group-by are very important; they fully determine which parts of the data are lifted to collections after the group-by operation.

An order-by qualifier works in the same way as a group-by qualifier but it produces a sorted result (a list). If there is a following group-by qualifier or a generator over a bag, then the order is lost because the list becomes a bag. The special parametric type Inv(t), which has a single data constructor inv(v) for a value v of type t, inverts the total order of a t value from $\leq$ to $\geq$. For example,

$$\{\, (x,y) \mid (x,y) \in S,\ \textbf{order by}\ (\text{inv}(x), y) \,\}$$

orders $(x,y) \in S$ by major order x (descending) and minor order y (ascending). Here, x and y are not lifted to bags because they are pattern variables in $(\text{inv}(x), y)$.

7.1 Translation of Monoid Comprehensions to the Monoid Algebra

Group-by and order-by qualifiers may appear in multiple places in the same comprehension as well as in nested comprehensions. For all these cases, only the pattern variables that precede the group-by or order-by qualifier in the same comprehension must be lifted to collections, and if multiple group-by and order-by qualifiers exist, these variables will have to be lifted multiple times to nested collections. These transformations can be easier expressed if they are done in two steps. The first step is to eliminate all group-bys and order-bys from comprehensions by looking at each comprehension in its entirety, so that only the preceding pattern variables are lifted to collections. The second step is to translate

$$\frac{\Gamma \vdash e : \mathbb{T}_{\otimes}(t) \quad t = \mathbb{T}_{\oplus} \quad \otimes \preceq \oplus}{\Gamma \vdash \oplus/e : t} \tag{20a}$$

$$\frac{\Gamma \vdash e' : t}{\Gamma \vdash \{\, e' \mid \,\} : [t]} \tag{20b}$$

$$\frac{\Gamma \vdash e : \text{boolean} \quad \Gamma \vdash \{\, e' \mid \overline{q} \,\} : \mathbb{T}_{\otimes}(t)}{\Gamma \vdash \{\, e' \mid e, \overline{q} \,\} : \mathbb{T}_{\otimes}(t)} \tag{20c}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \mathbb{T}_{\otimes}(t) \\ \Gamma, p/t \vdash \{\, e' \mid \overline{q} \,\} : \mathbb{T}_{\oplus}(t')\end{array}}{\Gamma \vdash \{\, e' \mid p \in e, \overline{q} \,\} : \mathbb{T}_{\max(\oplus,\otimes)}(t')} \tag{20d}$$

$$\frac{\Gamma \vdash e : t \quad \Gamma, p/t \vdash \{\, e' \mid \overline{q} \,\} : \mathbb{T}_{\otimes}(t')}{\Gamma \vdash \{\, e' \mid \textbf{let}\ p = e, \overline{q} \,\} : \mathbb{T}_{\otimes}(t')} \tag{20e}$$

Fig. 5. Type Rules for Monoid Comprehensions

the resulting comprehensions to the monoid algebra by translating each qualifier in a comprehension from left to right.

The first step is to translate the group-by and order-by qualifiers to groupBy and orderBy operations, respectively. We use the notation $\overline{q}$ to represent a sequence of qualifiers in a comprehension. Let $\mathcal{V}_{\overline{q}}^{p}$ be a flat tuple that contains all pattern variables in $\overline{q}$ that do not appear in p, defined as follows (the order of $v_1, \ldots, v_n$ is unimportant):

$$\begin{array}{rcll}
\mathcal{V}_{\overline{q}}^{p} & = & (v_1, \ldots, v_n) & \quad \text{where } v_i \in (\mathcal{V}[\![\overline{q}]\!] - \mathscr{P}[\![p]\!]) \\
\mathcal{V}[\![p \in e, \overline{q}]\!] & = & \mathscr{P}[\![p]\!] \cup \mathcal{V}[\![\overline{q}]\!] & \\
\mathcal{V}[\![\textbf{let}\ p = e, \overline{q}]\!] & = & \mathscr{P}[\![p]\!] \cup \mathcal{V}[\![\overline{q}]\!] & \\
\mathcal{V}[\![\textbf{group by}\, p : e, \overline{q}]\!] & = & \mathscr{P}[\![p]\!] \cup \mathcal{V}[\![\overline{q}]\!] & \\
\mathcal{V}[\![\textbf{order by}\, p : e, \overline{q}]\!] & = & \mathscr{P}[\![p]\!] \cup \mathcal{V}[\![\overline{q}]\!] & \\
\mathcal{V}[\![e, \overline{q}]\!] & = & \mathcal{V}[\![\overline{q}]\!] & \\
\mathscr{P}[\![(p_1, \ldots, p_n)]\!] & = & \mathscr{P}[\![p_1]\!] \cup \cdots \cup \mathscr{P}[\![p_n]\!] & \\
\mathscr{P}[\![v]\!] & = & \{v\} &
\end{array}$$

Then, group-by and order-by qualifiers can be eliminated from a monoid comprehension using the following rules:

$$\begin{array}{rl}
\{\, e' \mid \overline{q_1}, \textbf{group by}\, p : e, \overline{q_2} \,\} \rightarrow & \{\, e' \mid (p, s) \in \text{groupBy}(\{\, (e, \mathcal{V}_{\overline{q_1}}^{p}) \mid \overline{q_1} \,\}), \\
& \quad \forall v \in \mathcal{V}_{\overline{q_1}}^{p} :\ \textbf{let}\ v = \{\, v \mid \mathcal{V}_{\overline{q_1}}^{p} \in s \,\},\ \overline{q_2} \,\}
\end{array} \tag{19a}$$

$$\begin{array}{rl}
\{\, e' \mid \overline{q_1}, \textbf{order by}\, p : e, \overline{q_2} \,\} \rightarrow & \{\, e' \mid (p, s) \in \text{orderBy}(\{\, (e, \mathcal{V}_{\overline{q_1}}^{p}) \mid \overline{q_1} \,\}), \\
& \quad \forall v \in \mathcal{V}_{\overline{q_1}}^{p} :\ \textbf{let}\ v = \{\, v \mid \mathcal{V}_{\overline{q_1}}^{p} \in s \,\},\ \overline{q_2} \,\}
\end{array} \tag{19b}$$

Here, $\forall v \in \mathcal{V}_{\overline{q_1}}^{p} :\ \textbf{let}\ v = \{\, v \mid \mathcal{V}_{\overline{q_1}}^{p} \in s \,\}$ embeds a let-binding for each variable v in $\mathcal{V}_{\overline{q_1}}^{p}$ so that this variable is lifted to a collection that contains all v values in the current group. (The monoid of this collection is the maximum monoid in $\overline{q_1}$.) Note that, if e is missing in **group by** $p : e$ or **order by** $p : e$, then e is set to be equal to p, which means that the pattern variables in p may have been defined in $\overline{q_1}$. This is the case that we need to subtract $\mathscr{P}[\![p]\!]$ from $\mathcal{V}[\![\overline{q_1}]\!]$ in $\mathcal{V}_{\overline{q_1}}^{p}$, because the variables in p are group-by variables and should not be lifted to collections. Theorem A.5 in the Appendix shows that Rules (19a) and (19b) can be applied to a comprehension with multiple group-by and order-by qualifiers in any order.

The type rules for monoid comprehensions without group-by or order-by qualifiers are shown in Fig. 5. We use the notation $\Gamma, p/t$, for a pattern p, to bind the pattern variables in p to types. Eq. (20d) indicates that the collection monoid of a comprehension with a generator is $\max(\oplus,\otimes)$. That is, the monoid of a comprehension with multiple generators is the maximum monoid of all generator domains in the comprehension, starting with the monoid $+\!\!+$ for the empty comprehension (Eq. (20b)).

Comprehensions with group-by and order-by qualifiers can be type-checked if we consider each comprehension in its entirety, before the type rules in Fig. 5 are applied:

$$\frac{\begin{array}{c}\Gamma \vdash \{\, (e, \mathscr{V}^{p}_{\overline{q_1}}) \mid \overline{q_1} \,\} : \mathbb{T}_{\oplus}(t_0 \times (t_1 \times \ldots \times t_n)) \\ \Gamma, p/t_0, \mathscr{V}^{p}_{\overline{q_1}}/(\mathbb{T}_{\oplus}(t_1) \times \cdots \times \mathbb{T}_{\oplus}(t_n)) \vdash \{\, e' \mid \overline{q_2} \,\} : \mathbb{T}_{\otimes}(t)\end{array}}{\Gamma \vdash \{\, e' \mid \overline{q_1}, \textbf{group by}\, p : e, \overline{q_2} \,\} : \mathbb{T}_{\max(\uplus,\otimes)}(t)} \tag{21a}$$

$$\frac{\begin{array}{c}\Gamma \vdash \{\, (e, \mathscr{V}^{p}_{\overline{q_1}}) \mid \overline{q_1} \,\} : \mathbb{T}_{\oplus}(t_0 \times (t_1 \times \ldots \times t_n)) \\ \Gamma, p/t_0, \mathscr{V}^{p}_{\overline{q_1}}/(\mathbb{T}_{\oplus}(t_1) \times \cdots \times \mathbb{T}_{\oplus}(t_n)) \vdash \{\, e' \mid \overline{q_2} \,\} : \mathbb{T}_{\otimes}(t)\end{array}}{\Gamma \vdash \{\, e' \mid \overline{q_1}, \textbf{order by}\, p : e, \overline{q_2} \,\} : \mathbb{T}_{\otimes}(t)} \tag{21b}$$

The only difference between Eqs. (21a) and (21b) is in the monoid of the resulting type, which is $\max(\uplus,\otimes)$ for group-by and $\max(+\!\!+,\otimes) = \otimes$ for order-by. In both cases, each pattern variable v_i in $\overline{q_1}$ of type t_i is lifted to a collection of type $\mathbb{T}_{\oplus}(t_i)$.

The second step is to translate comprehensions without group-by or order-by qualifiers to the monoid algebra. We tag each comprehension with a collection monoid, as $\{\, e \mid \overline{q} \,\}_{\oplus}$, to enforce a minimum collection monoid for the comprehension. Initially, the minimum monoid is $+\!\!+$ to indicate there is no minimum restriction. The translation rules, which are similar to the translation rules for monad comprehensions (Wadler, 1987), are:

$$\oplus/e \rightarrow \text{reduce}(\oplus, e) \tag{22a}$$

$$\{\, e' \mid p \in e, \overline{q} \,\}_{\oplus} \rightarrow \text{cMap}(\lambda p.\{\, e' \mid \overline{q} \,\}_{\max(\oplus,\otimes)}, e) \qquad \text{where } e : \mathbb{T}_{\otimes}(t) \tag{22b}$$

$$\{\, e' \mid \textbf{let}\ p = e, \overline{q} \,\}_{\oplus} \rightarrow \textbf{let}\ p = e\ \textbf{in}\ \{\, e' \mid \overline{q} \,\}_{\oplus} \tag{22c}$$

$$\{\, e' \mid e, \overline{q} \,\}_{\oplus} \rightarrow \textbf{if}\ e\ \textbf{then}\ \{\, e' \mid \overline{q} \,\}_{\oplus}\ \textbf{else}\ \mathbf{1}_{\oplus} \tag{22d}$$

$$\{\, e' \mid \,\}_{\oplus} \rightarrow \mathbb{U}_{\oplus}(e') \tag{22e}$$

When we translate a generator in a comprehension in Eq. (22b), the minimum collection monoid $\oplus$ of the comprehension is refined to $\max(\oplus,\otimes)$, where $\otimes$ is the collection monoid of the generator domain. That way, the unit in Eq. (22e), which determines the final monoid of the comprehension, will be the maximum monoid of all generator domains in the comprehension.

For example, consider this monoid comprehension for a list X of type [int × int]:

$$\{\, (a,c) \mid (a,b) \in X,\ \ \textbf{order by}\ a,\ \ c \in b \,\}$$

Here, the order-by expression e in Eq. (21b) is equal to the order-by pattern a. In addition, $\mathscr{V}^{p}_{\overline{q_1}} = b$. Hence, $\{\, (e, \mathscr{V}^{p}_{\overline{q_1}}) \mid \overline{q_1} \,\}$ in Eq. (21b) is equal to $\{\, (a,b) \mid (a,b) \in X \,\}$, which has type [int × int]. Therefore, the type of the order-by comprehension is equal to the type of $\{\, (a,c) \mid c \in b \,\}$ under the type bindings a : int and b : [int], which is [int × int]. This comprehension is translated to the monoid algebra using Eq. (19b) as follows:

$$\begin{aligned}
&\{\, (a,c) \mid (a,s) \in \text{orderBy}(\{\, (a,b) \mid (a,b) \in X \,\}),\ \textbf{let}\ b = \{\, b \mid b \in s \,\},\ c \in b \,\}\\
&= \{\, (a,c) \mid (a,s) \in \text{orderBy}(\{\, (a,b) \mid (a,b) \in X \,\}),\ c \in s \,\}\\
&= \text{cMap}(\lambda(a,s).\, \text{cMap}(\lambda c.\, [(a,c)],\, s),\\
&\qquad\qquad \text{orderBy}(\text{cMap}(\lambda(a,b).\, \{\!\{(a,b)\}\!\}, X)))
\end{aligned}$$

If X were a bag, then the previous comprehension would have the type $\{\!\{\text{int}\times\text{int}\}\!\}$, since the type bindings are $a : \text{int}$ and $b : \{\!\{\text{int}\}\!\}$, which means that the order is lost. The comprehension $\{\, (a,b) \mid (a,b) \in X,\ \ \textbf{order by}\ a \,\}$, though, returns a list of type $[\text{int}\times\{\!\{\text{int}\}\!\}]$ that preserves the order.

As another example, matrix multiplication is translated to the monoid algebra as follows:

$$\begin{aligned}
&\{\, (+/z, i, j) \mid (x,i,k) \in X,\ (y,k',j) \in Y,\ k = k',\ \textbf{let}\ z = x*y,\ \textbf{group by}\ (i,j) \,\}\\
&= \{\, (\text{reduce}(+,z), i, j) \mid ((i,j),s) \in \text{groupBy}(\{\, ((i,j),(x,k,y,k',z))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid (x,i,k) \in X,\ (y,k',j) \in Y,\ k = k',\ \textbf{let}\ z = x*y \,\}),\\
&\qquad\qquad \textbf{let}\ x = \{\, x \mid (x,k,y,k',z) \in s \,\},\ \ldots,\ \textbf{let}\ z = \{\, z \mid (x,k,y,k',z) \in s \,\}\,\}\\
&= \{\, (\text{reduce}(+, \{\, z \mid (x,k,y,k',z) \in s \,\}), i, j)\\
&\quad \mid ((i,j),s) \in \text{groupBy}(\{\, ((i,j),(x,k,y,k',x*y))\\
&\qquad\qquad\qquad \mid (x,i,k) \in X,\ (y,k',j) \in Y,\ k = k' \,\})\,\}\\
&= \text{cMap}(\lambda((i,j),s).\, \{\!\{(\text{reduce}(+, \text{cMap}(\lambda(x,k,y,k',z).\, \{\!\{z\}\!\}, s)), i, j)\}\!\},\\
&\qquad \text{groupBy}(\text{cMap}(\lambda(x,i,k).\, \text{cMap}(\lambda(y,k',j).\\
&\qquad\qquad \textbf{if}\ k = k'\ \textbf{then}\ \{\!\{((i,j),(x,k,y,k',x*y))\}\!\}\ \textbf{else}\ \{\!\{\,\}\!\}, Y), X)))
\end{aligned}$$

Monoid comprehensions without group-by or order-by qualifiers can be normalized using the well-known law on comprehensions:

$$\{\, e \mid \overline{q_1},\, p \in \{\, e' \mid \overline{q_3} \,\},\, \overline{q_2} \,\} \rightarrow \{\, e \mid \overline{q_1},\, \overline{q_3},\, \textbf{let}\ p = e',\, \overline{q_2} \,\} \tag{23}$$

after renaming the variables in $\{\, e' \mid \overline{q_3} \,\}$ to prevent variable capture. We prove this law in Theorem A.6 in the Appendix.

8 MRQL: A Query Language for Big Data Analytics

Data analysts and database users are more familiar with SQL-like syntax than comprehensions. Our monoid algebra has been used as the basis for Apache MRQL (Apache MRQL, 2017), which is a query processing and optimization system for large-scale, distributed data analysis. MRQL was originally developed by the author (Fegaras *et al.*, 2012), but is now an Apache incubating project with many developers and users worldwide. The MRQL language is an SQL-like query language for large-scale data analysis on computer clusters. The MRQL query processing system can evaluate MRQL queries in four modes: in Map-Reduce mode using Apache Hadoop, in BSP (Bulk Synchronous Parallel) mode using Apache Hama, in Spark mode using Apache Spark, and in Flink mode using Apache Flink. The MRQL query language is powerful enough to express most common data analysis

$$\begin{array}{rrcll}
\textbf{types:} & t & ::= & \text{basic} \mid \text{named-type} \mid \text{bag}(t) \mid \text{list}(t) \mid (t_1,\ldots,t_n) \mid \langle A_1:t_1,\ldots,A_n:t_n\rangle & \\
\textbf{patterns:} & p & ::= & v \mid * \mid (p_1,\ldots,p_n) \mid \langle A_1:p_1,\ldots,A_n:p_n\rangle & \\
\textbf{qualifiers:} & q & ::= & p\ \textbf{in}\ e \mid p = e & \\
\textbf{expressions:} & e & ::= & \textbf{select}\ [\textbf{distinct}]\ e'\ \textbf{from}\ q_1,\ldots,q_n & \text{// select query}\\
&&& \quad [\textbf{where}\ e_p] & \\
&&& \quad [\textbf{group by}\ p\,[:e]\ \ [\textbf{having}\ e_h]] & \\
&&& \quad [\textbf{order by}\ p_o\,[:e_o]] & \\
&& \mid & \textbf{repeat}\ v = e\ \textbf{step}\ e'\ [\textbf{limit}\ n] & \text{// repetition}\\
&& \mid & \textbf{some}\ q_1,\ldots,q_n:\ e_p & \text{// existential quantification}\\
&& \mid & \textbf{all}\ q_1,\ldots,q_n:\ e_p & \text{// universal quantification}\\
&& \mid & e_1\ \textbf{union}\ e_2 & \text{// bag union}\\
&& \mid & \text{aggregation}(e) & \text{// named aggregation}\\
&& \mid & \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \ \mid\ \textbf{let}\ p = e\ \textbf{in}\ e' & \\
&& \mid & (e_1,\ldots,e_n) \mid \langle A_1:e_1,\ldots,A_n:e_n\rangle \mid e\#i \mid e.A & \\
&& \mid & \{e_1,\ldots,e_n\} \mid [e_1,\ldots,e_n] \mid f(e_1,\ldots,e_n) \mid e_1[e_2] \mid (e) \mid v \mid \mathit{const} \mid \ldots &
\end{array}$$

Fig. 6. The MRQL Syntax

tasks over many forms of raw in-situ data, such as XML and JSON documents, binary files, and CSV documents. The design of MRQL has been influenced by XQuery and ODMG OQL, although it uses SQL-like syntax. In fact, when restricted to XML, MRQL is as powerful as XQuery. With MRQL, users are able to express complex data analysis tasks, such as PageRank, k-means clustering, matrix factorization, etc, using SQL-like queries exclusively, while the MRQL query processing system is able to compile these queries to efficient Java code that can run on various DISC platforms.

The MRQL data model consists of lists, bags, records, tuples, algebraic data types (union types), parametric types, and basic types, such as integers and booleans. These types can be freely nested, thus supporting nested relations and hierarchical data. For example, XML data can be represented as a recursive algebraic data type with two data constructors, Node and CData:

```
data XML = Node: < tag: String, attributes: { (String, String) },
                   children :  list ( XML ) >
         | CData: String
```

It is interesting to see how MRQL processes instances of hierarchical data types, such as XML and JSON, in parallel. This actually addresses the main criticism against monoid homomorphisms, as they require data constructors to be associative. In distributed processing, each processing node is assigned a data split that consists of data fragments. Thus, hierarchical data, defined as algebraic data types, must be fragmented into manageable pieces, so that the algebraic data type T becomes a list of fragments, list(T). MRQL provides a customizable fragmentation technique to suit a wide range of application needs. The MRQL expression used for parsing an XML document is source(xml,path,tags,xpath), where path is the document path, tags is a bag of synchronization tags, and xpath is the XPath expression used for fragmentation. Given a data split from the XML document, this operation skips all text until it finds the opening of a synchronization tag and then stores the text upto the matching closing tag into a buffer. The buffer then becomes the

$$\textbf{general qualifiers:}\;\; Q \;::=\; q \mid \textbf{where}\, e \mid \textbf{group by}\, p\,[: e] \mid \textbf{having}\, e \mid \textbf{order by}\, p\,[: e]$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathscr{E}\;:\;\text{MRQL expression} \rightarrow \text{monoid comprehensions and algebraic operations}}\\
\mathscr{E}[\![\textbf{select distinct}\, e\, \textbf{from}\, Q_1 \ldots Q_n]\!] & = & \{\, x \mid x \in \mathscr{E}[\![\textbf{select}\, e\, \textbf{from}\, Q_1 \ldots Q_n]\!],\, \textbf{group by}\, x\,\}\\
\mathscr{E}[\![\textbf{select}\, e\, \textbf{from}\, Q_1 \ldots Q_n]\!] & = & \{\, e \mid \mathscr{Q}[\![Q_1]\!], \ldots, \mathscr{Q}[\![Q_n]\!]\,\}\\
\mathscr{E}[\![\textbf{some}\, q_1, \ldots, q_n :\, e]\!] & = & \vee/\{\, \mathscr{E}[\![e]\!] \mid \mathscr{Q}[\![q_1]\!], \ldots, \mathscr{Q}[\![q_n]\!]\,\}\\
\mathscr{E}[\![\textbf{all}\, q_1, \ldots, q_n :\, e]\!] & = & \wedge/\{\, \mathscr{E}[\![e]\!] \mid \mathscr{Q}[\![q_1]\!], \ldots, \mathscr{Q}[\![q_n]\!]\,\}\\
\mathscr{E}[\![\textbf{repeat}\, v = e\, \textbf{step}\, e'\, \textbf{limit}\, n]\!] & = & \text{repeat}(\lambda v.\mathscr{E}[\![e']\!],\, \mathscr{E}[\![e]\!],\, \mathscr{E}[\![n]\!])\\
\mathscr{E}[\![\text{aggregation}(e)]\!] & = & \oplus/\mathscr{E}[\![e]\!] \qquad \text{where } \oplus \text{ is the aggregation monoid}
\end{array}$$

$$\begin{array}{lcl}
\multicolumn{3}{l}{\mathscr{Q}\;:\;\text{general qualifier} \rightarrow \text{monoid comprehension qualifier}}\\
\mathscr{Q}[\![p\, \textbf{in}\, e]\!] & = & p \in \mathscr{E}[\![e]\!]\\
\mathscr{Q}[\![p = e]\!] & = & \textbf{let}\, p = \mathscr{E}[\![e]\!]\\
\mathscr{Q}[\![\textbf{where}\, e]\!] & = & \mathscr{E}[\![e]\!]\\
\mathscr{Q}[\![\textbf{group by}\, p\,[: e]\,]\!] & = & \textbf{group by}\, p\,[: \mathscr{E}[\![e]\!]\,]\\
\mathscr{Q}[\![\textbf{having}\, e]\!] & = & \mathscr{E}[\![e]\!]\\
\mathscr{Q}[\![\textbf{order by}\, p\,[: e]\,]\!] & = & \textbf{order by}\, p\,[: \mathscr{E}[\![e]\!]\,]
\end{array}$$

Fig. 7. Some Translation Rules from MRQL to Monoid Comprehensions

current context for xpath, which returns a sequence of XML objects. The XML processing framework for MRQL is described in (Fegaras *et al.*, 2011).

The MRQL syntax is defined in Fig. 6. Note that the curly bracket syntax, $\{e_1, \ldots, e_n\}$, constructs a bag, not a set, since sets are not supported in MRQL. The MRQL syntax is translated to monoid comprehensions using the transformations in Fig. 7. Without loss of generality, to simplify the translation rules, we have taken the MRQL query parts after the 'from' keyword to be general qualifiers, although the order they can appear in an MRQL query can only be the order shown in Fig. 6. For example, matrix multiplication can be translated as follows:

$$\begin{aligned}
&\mathscr{E}[\![\textbf{select}\, (\textbf{sum}(z), i, j)\, \textbf{from}\, (x, i, k)\, \textbf{in}\, X,\; (y, k', j)\, \textbf{in}\, Y,\; z = x * y\\
&\qquad \textbf{where}\, k = k'\, \textbf{group by}\, (i, j)]\!]\\
&= \{\, (\textbf{sum}(z), i, j) \mid \mathscr{Q}[\![(x, i, k)\, \textbf{in}\, X]\!], \mathscr{Q}[\![(y, k', j)\, \textbf{in}\, Y]\!], \mathscr{Q}[\![z = x * y]\!],\\
&\qquad\qquad\qquad\qquad \mathscr{Q}[\![\textbf{where}\, k = k']\!], \mathscr{Q}[\![\textbf{group by}\, (i, j)]\!]\,\}\\
&= \{\, (+/z, i, j) \mid (x, i, k) \in X,\, (y, k', j) \in Y,\, \textbf{let}\, z = x * y,\, k = k',\, \textbf{group by}\, (i, j)\,\}
\end{aligned}$$

Fig. 8 gives some of the type rules for the MRQL select queries. They are directly derived from the type rules for monoid comprehensions. Some select queries in the type rules use regular quantifiers $\overline{q}$ while others use general qualifiers $\overline{Q}$ so that those with general qualifiers are matched first. The parametric types T, T_1, and T_2 in Fig. 8 are collection types (lists or bags). The max function in Eq. (24c) returns the max collection type, based on the order list $\prec$ bag.

MRQL supports a number of predefined aggregation functions, such as count and sum, but it also provides syntax to define new aggregations as monoids:

```
aggregation aggr_name ( plus, zero ) : t
```

$$\frac{\begin{array}{c}\Gamma \vdash (\textbf{select } (e, \mathcal{V}_{\overline{Q}}^{p}) \textbf{ from } \overline{Q}) : T((t_0, (t_1, \ldots, t_n))) \\ \Gamma, p/t_0, \mathcal{V}_{\overline{Q}}^{p}/(T(t_1), \ldots, T(t_n)) \vdash e' : t\end{array}}{\Gamma \vdash (\textbf{select } e' \textbf{ from } \overline{Q} \textbf{ group by } p : e) : \text{bag}(t)} \tag{24a}$$

$$\frac{\begin{array}{c}\Gamma \vdash (\textbf{select } (e, \mathcal{V}_{\overline{Q}}^{p}) \textbf{ from } \overline{Q}) : T((t_0, (t_1, \ldots, t_n))) \\ \Gamma, p/t_0, \mathcal{V}_{\overline{Q}}^{p}/(T(t_1), \ldots, T(t_n)) \vdash e' : t\end{array}}{\Gamma \vdash (\textbf{select } e' \textbf{ from } \overline{Q} \textbf{ order by } p : e) : \text{list}(t)} \tag{24b}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_1(t) \\ \Gamma, p/t \vdash (\textbf{select } e' \textbf{ from } \overline{q}) : T_2(t') \\ T = \max(T_1, T_2)\end{array}}{\Gamma \vdash (\textbf{select } e' \textbf{ from } p \textbf{ in } e,\ \overline{q}) : T(t')} \tag{24c}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \text{boolean} \\ \Gamma \vdash (\textbf{select } e' \textbf{ from } \overline{Q}) : T(t)\end{array}}{\Gamma \vdash (\textbf{select } e' \textbf{ from } \overline{Q} \textbf{ having } e) : T(t)} \tag{24e}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \\ \Gamma, p/t \vdash (\textbf{select } e' \textbf{ from } \overline{q}) : T(t')\end{array}}{\Gamma \vdash (\textbf{select } e' \textbf{ from } p = e,\ \overline{q}) : T(t')} \tag{24d}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \text{boolean} \\ \Gamma \vdash (\textbf{select } e' \textbf{ from } \overline{q}) : T(t)\end{array}}{\Gamma \vdash (\textbf{select } e' \textbf{ from } \overline{q} \textbf{ where } e) : T(t)} \tag{24f}$$

Fig. 8. Some Type Rules for MRQL

where aggr_name is the name of the aggregation, t is the type, plus is an associative function of type $(t,t) \to t$, and zero is the identity of plus. Then, aggr_name(e) is translated to reduce(plus, e). The MRQL system will type-check the monoid components of this definition but will not enforce the monoid properties.

9 Converting Nested cMaps to Joins

The translation rules from comprehensions to the monoid algebra, presented in Section 7.1, do not generate coGroup operations because these rules do not include cases that match joins, such as $\{\, (x,y) \mid x \in X,\, y \in Y,\, p(x,y) \,\}$, for some join predicate $p(x,y)$. Such comprehensions with multiple generators were translated to nested cMaps. In this section, we present a general method for identifying any possible equi-join (a join whose join predicate takes the form $k_1(x) = k_2(y)$, for some key function k_1 and k_2), including joins across deeply nested queries. It is exactly because of these deeply nested queries that we have introduced the coGroup operation, because, as we will see, nested queries over bags are equivalent to outer joins. Translating nested cMaps to coGroups is crucial for good performance in distributed processing. The only way to evaluate a nested cMap, such as $\text{cMap}(\lambda x.\, \text{cMap}(\lambda y.\, h(x,y),\, Y),\, X)$, in a distributed environment, where both collections X and Y are distributed, is to broadcast the entire collection Y across the processing nodes so that each processing node would join its own partition of X with the entire dataset Y. This is a good evaluation plan if Y is small. By mapping a nested cMap to a coGroup, we create opportunities for more evaluation strategies, which may include the broadcast evaluation. For example, one good evaluation strategy for large X and Y joined via the key functions k_1 and k_2, is to partition X by k_1, partition Y by k_2, and shuffle these partitions to the processing nodes so that data with matching keys will go to the same processing node. This is called a distributed partitioned join. While related approaches for query unnesting (Fegaras and Maier, 2000; Holsch *et al.*, 2016) require many rewrite rules to handle various cases of query nesting, our method requires only one and is more general as it handles nested queries of any form and any number of nesting levels.

Consider the following nested query over the bags X and Y:

```
select x from x in X
where x.D > sum( select y.C from y in Y where x.A=y.B )
```

which is equivalent to the comprehension:

$$\{\, x \mid x \in X,\; x.D > +/\{\, y.C \mid y \in Y, x.A = y.B \,\}\,\}$$

A typical method for evaluating this nested query in a relational database system is to first group Y by y.B:

```
select y.B, sum(y.C) from y in Y group by y.B
```

and then to join the result of this group-by query with X on x.A=y.B using a left-outer join, keeping all those matches for which x.D is less than the sum. The outer join is necessary because, otherwise, we may dismiss an x that has no matches in Y with x.A=y.B. Using the monoid algebra, we want to translate this query to:

$$\begin{array}{l}\text{cMap}(\lambda(k,(xs,ys)).\,\text{cMap}(\lambda x.\,\textbf{if}\; x.D > \text{reduce}(+,ys)\;\textbf{then}\;\{\!\{x\}\!\}\;\textbf{else}\;\{\!\{\,\}\!\},\; xs)\\ \qquad\quad \text{coGroup}(\text{cMap}(\lambda x.\,\{\!\{(x.A,x)\}\!\}, X),\\ \qquad\qquad\qquad \text{cMap}(\lambda y.\,\{\!\{(y.B,y.C)\}\!\}, Y)))\end{array}$$

That is, the query unnesting is done with a left-outer join, which is captured concisely by the coGroup operation without the need for using an additional group-by operation or handling null values. We generalize this unnesting technique to convert pairs of nested cMaps to joins.

We consider patterns of algebraic terms of the form

$$\text{cMap}(\lambda x.\, g(\text{cMap}(\lambda y.\, h(x,y),\, e_2)), e_1)$$

for some terms e_1 and e_2 and some term functions g and h (i.e., terms that contain their arguments as subterms). This pattern is equivalent to the monoid comprehension

$$\{\, z \mid x \in e_1, z \in g(\{\, w \mid y \in e_2, w \in h(x,y)\,\})\,\}$$

For the cases we consider, the term e_2 should not depend on x, otherwise it would not be a join, and the terms e_1 and e_2 should not be lists, otherwise the derived join may destroy the list order. This pattern matches any pair of nested cMaps on bags, including those derived from nested queries, such as the previous MRQL query, and those derived from join-like comprehensions, such as $\{\, e' \mid \ldots, x \in e_1, \ldots, y \in e_2, \ldots, k_1(x) = k_2(y), \ldots \,\}$. Thus, the method presented here detects and converts any possible join to a coGroup. But, it may also match a term in multiple ways, such as the term derived from $\{\, e \mid x \in X, y \in Y, z \in Z, \ldots \,\}$, which is translated to a triple-nested cMap. In that case, any match order can be used, because the derived coGroups can be optimized further by using cost-based rewrite rules that exploit the associativity and commutativity properties of coGroup on bags.

Let $F(X,Y) = \text{cMap}(\lambda x.\, g(\text{cMap}(\lambda y.\, h(x,y), Y)), X)$. To transform this term to a join, we need to derive a join predicate from $h(x,y)$. More specifically, we need to derive two join key functions k_1 and k_2 such that $k_1(x) \neq k_2(y)$ implies $h(x,y) = \{\!\{\,\}\!\}$. This is equivalent to the condition $h(x,y) = \{\, z \mid k_1(x) = k_2(y),\, z \in h(x,y) \,\}$. Then, if there are such functions k_1 and k_2, we can do the following transformation on $F(X,Y)$:

$$\begin{aligned} F(X,Y) \;\rightarrow\; & \text{cMap}(\lambda(k,(xs,ys)).\,F(xs,ys), \\ & \qquad \text{coGroup}(\text{cMap}(\lambda x.\,\{\!\{(k_1(x),x)\}\!\}, X), \\ & \qquad\qquad \text{cMap}(\lambda y.\,\{\!\{(k_2(y),y)\}\!\}, Y))) \end{aligned} \tag{25}$$

This law is proven in Theorem A.7 in the Appendix. This is the only transformation rule needed to derive any possible join from a query and unnest nested queries.

For example, $\{\, a \mid (a,b) \in X,\, a > +/\{\, c \mid (b',c) \in Y,\, b' = b \,\}\,\}$, which is translated to

$$\begin{aligned} & \text{cMap}(\lambda(a,b).\,\textbf{if}\ a > \text{reduce}(+, \text{cMap}(\lambda(b',c).\,\textbf{if}\ b' = b\ \textbf{then}\ \{\!\{c\}\!\}\ \textbf{else}\ \{\!\{\,\}\!\}, Y)) \\ & \qquad\qquad \textbf{then}\ \{\!\{a\}\!\}\ \textbf{else}\ \{\!\{\,\}\!\}, X) \end{aligned}$$

is transformed to

$$\begin{aligned} & \text{cMap}(\lambda(k,(xs,ys)).\,\text{cMap}(\lambda(a,b).\,\textbf{if}\ a > \text{reduce}(+, \text{cMap}(\lambda(b',c).\,\{\!\{c\}\!\}, ys)) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ \{\!\{a\}\!\}\ \textbf{else}\ \{\!\{\,\}\!\}, xs), \\ & \qquad \text{coGroup}(\text{cMap}(\lambda(a,b).\,\{\!\{(b,(a,b))\}\!\}, X), \\ & \qquad\qquad \text{cMap}(\lambda(b,c).\,\{\!\{(b,(b,c))\}\!\}, Y))) \end{aligned}$$

where the cMap inside the reduce was simplified, given that $b' = b$.

The main challenge in applying this transformation is to derive the join key functions, k_1 and k_2, from the term function h. If cMaps are normalized first using Eq. (6), then the only place these key functions can appear is in a cMap functional argument. Then, the body of the cMap functional argument can be another cMap, in which case we consider the latter cMap, or a term **if** $p(x,y)$ **then** e **else** $\{\!\{\,\}\!\}$, in which case we derive the keys from $p(x,y)$, such that $k_1(x) \neq k_2(y) \Rightarrow \neg p(x,y)$. If $p(x,y)$ is already in the form $k_1(x) = k_2(y)$, then in addition to deriving the keys k_1 and k_2, we can simplify the term $F(xs,ys)$ in Eq. (25) by replacing the term **if** $p(x,y)$ **then** e **else** $\{\!\{\,\}\!\}$ with e, as we did in the nested query example.

10 Other Optimizations

Based on the naive implementation of the monoid algebraic operators given in Section 5, the most expensive operation would be the coGroup, followed by the groupBy, since they both require shuffling the data across the processing nodes. The cMap operation, on the other hand, does not require any data shuffling, and can be performed in parallel at each processing node. Therefore, it is important to minimize the number of groupBy and coGroup operations and convert coGroups to groupBys, when possible.

First, consider a very important optimization: translating a self-join to a single group-by. Self-joins are actually very common in data analysis queries, especially in graph analysis. Many graph algorithms are repetitive self-joins over the graph, following a breadth-first search pattern. For example, the PageRank algorithm computes the importance of the web pages in a web graph based exclusively on the topology of the graph. For a graph with vertices V and edges E, the PageRank P_i of a vertex $v_i \in V$ is calculated from the PageRank P_j of its incoming neighbors $v_j \in V$ with $(v_j, v_i) \in E$ using the recursive rule $P_i =$

$\sum_{(v_j,v_i)\in E} \frac{P_j}{\|\{v_k\,|\,(v_j,v_k)\in E\}\|}$. If we represent a vertex v_i as a record of type $\langle$ id: long, adjacent: bag(long) $\rangle$, where adjacent contains the destination vertices v_j, such that $(v_i,v_j)\in E$, then the following MRQL query computes one step of the PageRank algorithm over the web graph Graph:

```
select < id: n.id, rank: n.rank, adjacent: m.adjacent >
  from n in (select < id: a, rank: sum(in_rank) >
                from n in Graph
                     a in n.adjacent,
                     in_rank = n.rank/count(n.adjacent)
               group by a),
         m in Graph
 where m.id = n.id
```

The inner select query calculates the new PageRanks while the outer select query reconstructs the graph to prepare it for the next repetition step. If it is implemented naively in a DISC platform, the algebraic term of this query would require two stages, one for the group-by and another for the self-join. But here, not only the join is a self-join, but also the join key is equal to the group-by key. It is well-known though that a PageRank step can be computed using a single stage (Lin and Dyer, 2010). Consequently, we would like to derive an algebraic term that contains one groupBy only.

The first transformation applies when a coGroup is over a groupBy and the groupBy key is the same as the join key. Then the groupBy can be eliminated since the coGroup is equivalent to a groupBy on both inputs:

$$\begin{aligned}&\text{coGroup}(\text{cMap}(\lambda(k,s).\,\{\!\{(k,f(s))\}\!\},\,\text{groupBy}(X)),Y)\\ &\;\rightarrow\;\text{cMap}(\lambda(k,(xs,ys)).\,\{\!\{(k,(\{\!\{f(xs)\}\!\},ys))\}\!\},\,\text{coGroup}(X,Y))\end{aligned} \tag{26}$$

The correctness of this transformation is proven in Theorem A.8 in the Appendix. The second transformation converts a coGroup over the same input bag into a groupBy:

$$\begin{aligned}&\text{coGroup}(\text{cMap}(f,X),\,\text{cMap}(g,X))\\ &\rightarrow\text{cMap}(\lambda(k,s).\,\{\!\{(k,(\text{cMap}(\lambda v.\,\textbf{case}\;v\;\textbf{of}\;\textbf{inL}(a)\Rightarrow\{\!\{a\}\!\}\mid\textbf{inR}(b)\Rightarrow\{\!\{\,\}\!\},\,s),\\ &\qquad\qquad\qquad\qquad\text{cMap}(\lambda v.\,\textbf{case}\;v\;\textbf{of}\;\textbf{inL}(a)\Rightarrow\{\!\{\,\}\!\}\mid\textbf{inR}(b)\Rightarrow\{\!\{b\}\!\},\,s)))\}\!\},\\ &\qquad\quad\text{groupBy}(\text{cMap}(\lambda x.\,\text{cMap}(\lambda(k,a).\,\{\!\{(k,\textbf{inL}(a))\}\!\},\,f(x))\\ &\qquad\qquad\qquad\qquad\uplus\,\text{cMap}(\lambda(k,b).\,\{\!\{(k,\textbf{inR}(b))\}\!\},\,g(x)),\,X)))\end{aligned}$$

The correctness of this transformation can be proven using Theorem A.9 in the Appendix by substituting cMap(f,X) for X and cMap(g,X) for Y in the theorem. Note that neither of these transformation reduces the amount of the shuffled data; in the first transformation, the left coGroup input over the groupBy result is already shuffled on the join key, while the resulting groupBy in the second transformation shuffles the input values twice, as is done by the coGroup. Both transformations though reduce the time needed for barrier synchronization, since the first one eliminates one stage and the latter one converts a coGroup to a groupBy, which has less synchronization overhead. Based on these two transformations, the PageRank step is transformed to a term with one groupBy operation.

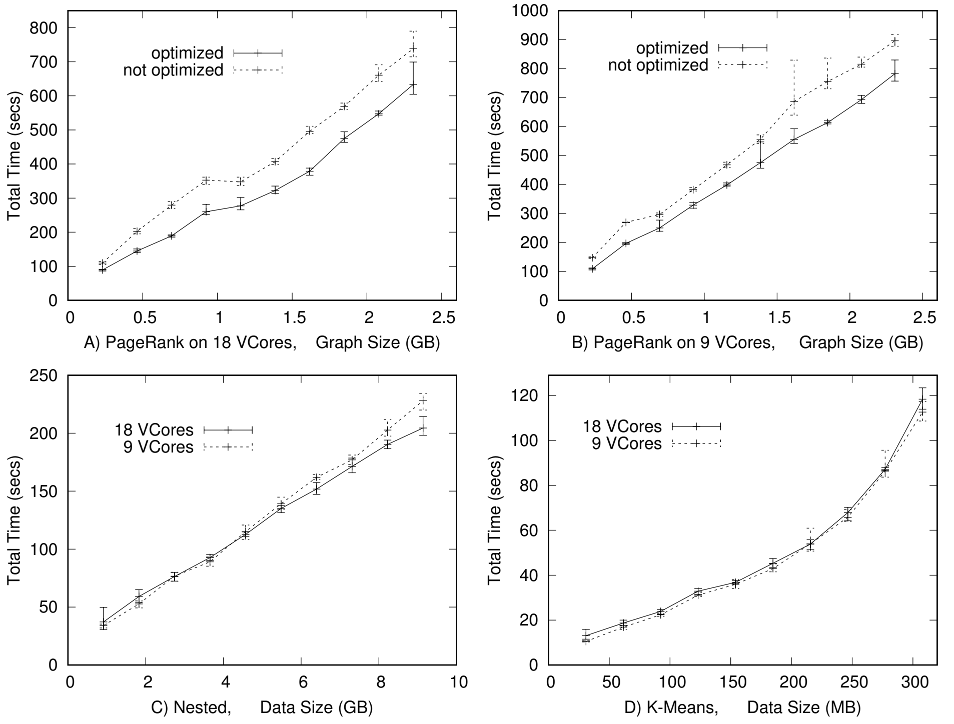

Fig. 9. Evaluation of PageRank, a Nested Query, and K-Means Clustering

The derived code is very similar to the code one may have written to implement the PageRank in a DISC framework using one stage per step.

11 Performance Evaluation

The platform used for our evaluations is a small cluster of 10 nodes, built on the Chameleon cloud computing infrastructure, `www.chameleoncloud.org`. Our virtual cluster consists of 10 m1.medium instances running Ubuntu Linux, each one with 4GB RAM, 40GB HDD, and 2 VCores at 2.3GHz. For our experiments, we used Apache Hadoop 2.6.0 (Yarn), Apache Flink 1.0.3, and Apache MRQL 0.9.6. The cluster frontend was used exclusively as a NameNode and ResourceManager, while the rest nine compute nodes were used as DataNodes and NodeManagers. There was a total of 18 VCores and a total of 36GB of RAM available for Flink tasks. The HDFS file system was formatted with the block size set to 128MB and the replication factor set to 3. Each dataset used in our experiments was stored as a single HDFS file. Our experiments were run on two different cluster sizes: one with 18 and another with 9 VCores. Each experiment was evaluated 6 times under the same data and configuration parameters. Each data point in the plots in Fig. 9 represent the mean value of 6 experiments while the vertical error bar at each line point represents the minimum and and maximum values among these 6 experiments.

Our first experiment is to evaluate the PageRank algorithm described in Section 10. To evaluate the effectiveness of the optimizations described in Section 10, we evaluated PageRank in two modes: with and without these optimizations. When these optimizations are used, the evaluation of PageRank requires only one stage per iteration, while, when these optimizations are turned off, the evaluation of PageRank requires two stages per iteration. The graphs used in our experiments were synthetic data generated by the RMAT (Recursive MATrix) Graph Generator (Chakrabarti *et al.*, 2011) using the Kronecker graph generator parameters a=0.30, b=0.25, c=0.25, and d=0.20. The number of distinct edges generated were 10 times the number of graph vertices. We used 10 datasets RMAT-*i*, for $i = 1 \ldots 10$, of size $i * 0.231$GB, with $i * 10^6$ vertices and $i * 10^7$ edges. That is, the largest dataset was 2.31GB. The PageRank repeat query used 10 iterations. The results of the PageRank evaluation for 18 VCores are shown in Fig. 9.A and for 9 VCores are shown in Fig. 9.B. With the optimizations turned on, the average speedup for 18 VCores was 28.6% and for 9 VCores was 22.1%.

The second experiment is to evaluate the nested query described in Section 9. Our system translates this query to a simple coGroup, which is implemented as a distributed partitioned join. Most other DISC query systems either do not permit such a query nesting, or, if they do, they evaluate this query as a cross product, by broadcasting the dataset used in the inner query to all the processing nodes. We did not evaluate the latter case because the broadcast dataset is too large to fit in the main memory of a processing node. We used 10 pairs of datasets X-*i* and Y-*i*, for $i = 1 \ldots 10$, where each dataset has $i * 2 * 10^7$ tuples and is of size $i * 0.914$GB. That is, the largest input has a total size 9.14GB. We used a many-to-many join with a maximum of $2 * 10$ matches for each join key. The results of the nested query evaluation are shown in Fig. 9.C. These results indicate that, when evaluating this nested query, there is little difference between 9 and 18 VCores; using more than 9 cores will not improve performance. This result may look surprising because one may expect that adding more cores will increase the degree of parallelism and hence decrease run-time, given that the amount of data shuffling during coGroup is independent of the number of cores. But when the number of cores increases, the cost for scheduling the data shuffling processes increases and may equalize the performance gain from increasing the degree of parallelism. In fact, there is an optimal number of cores for evaluating the groupBy and coGroup operations on data of certain size; beyond that number, performance may degrade.

The last experiment is to evaluate the following MRQL query that implements the k-means clustering algorithm by repeatedly deriving *k* new centroids from the old ones:

```
repeat centroids = ...
step select ( < X: avg(s.X), Y: avg(s.Y) >, true )
      from s in Points
      group by k: (select c from c in centroids
                        order by distance(c,s ))[0]
```

where Points is a dataset of points on the X-Y plane, centroids is the current set of centroids (*k* cluster centers), and distance is a function that calculates the Euclidean distance between two points. The initial value of centroids (the ... value) is a bag of *k* random points. The inner select-query in the group-by part assigns the closest centroid to a point s (where [0] returns the first tuple of an ordered list). The outer select-query in the repeat step clusters the data points by their closest centroid, and, for each cluster, a new centroid is calculated from the average values of its points. Most SQL-like DISC query systems do not allow nested queries in the group-by clause, but this time the naive evaluation plan, which is broadcasting the inner bag (the centroids), is the best strategy because there are only k centroids. The datasets used for the k-means query consist of random (X,Y) points in 4 squares that have X in $[2,4]$ or $[6,8]$ and Y in $[2,4]$ or $[6,8]$. Thus, the 4 centroids are expected to be $(3,3)$, $(3,7)$, $(7,3)$, and $(7,7)$. We used 10 datasets KMeans-i, for $i = 1\dots10$, where each datatset has $i*10^6$ points and is of size $i*30.8$MB. That is, the largest dataset is 308.1MB. The k-means query uses 10 iterations. The results of evaluating the k-means clustering query are shown in Fig. 9.D. As in the previous nested query, when evaluating the k-means query over these datasets, there is little difference between the results for 9 and 18 VCores. Like for the previous nested query, the explanation for this unexpected result is that the cost overhead of data shuffling equalizes the performance gain from increasing the degree of parallelism.

12 Current Work: Incrementalization

Recently, MRQL was extended to support the processing of continuous MRQL queries over streams of batch data (that is, data that come in continuous large batches). These extensions have been implemented on top of the Spark Streaming engine (Zaharia *et al.*, 2013). A more recent extension (Fegaras, 2016), called Incremental MRQL, extends the MRQL streaming engine with incremental stream processing capabilities to analyze data in incremental fashion, so that existing results on current data are reused and merged with the results of processing the new data. In many cases, incremental data processing can achieve better performance and may require less memory than batch processing for many common data analysis tasks. It can also be used for analyzing Big Data incrementally, in batches that can fit in memory, thus enabling us to process more data with less hardware. In addition, incremental data processing can be useful to stream-based applications that need to process continuous streams of data in real-time with low latency. There is a substantial body of work on incrementalization, including self-adjusting computation (Acar *et al.*, 2009), which translates batch programs into programs that can automatically respond to changes to their data. Our incremental processing framework is able to statically transform any batch MRQL query to an incremental stream processing query that returns accurate results, not approximate answers. This is accomplished by retaining a minimal state during the query evaluation lifetime and by returning an accurate snapshot answer at each time interval that depends on the current state and the latest batches of data.

The main idea of our approach is to convert an MRQL query q to a homomorphism so that $q(S \uplus \Delta S) = q(S) \otimes q(\Delta S)$, for some monoid $\otimes$, where S is the current data and ΔS is the new data. Then, we can maintain $q(S)$ as a state through streaming and combine it with $q(\Delta S)$ to derive a new state. Unfortunately, although our algebraic operators are homomorphic, their composition may not be (see Section 4.1). We have developed a general technique for transforming any algebraic term to a homomorphism. This is accomplished by lifting the term so that it propagates the groupBy and coGroup keys to the output. This is known as lineage tracking. That way, the query results are grouped by a key combination that corresponds the groupBy and coGroup keys used in deriving these values during query evaluation. If we also group the new data in the same way, then computations on existing data (the current state) can be combined with the computations on the new data by joining the data on these keys. In fact, the merging of the query result on the new data with the current state is done with $\Updownarrow_{\otimes}$, for some derived monoid $\otimes$. This full outer join is implemented efficiently as a distributed partitioned join, by keeping the state partitioned on the lineage keys and shuffling only the new results to processing nodes to be combined locally with the state using $\Updownarrow_{\otimes}$. The reader is referred to (Fegaras, 2016) for more details.

13 Conclusion

We have presented an algebra that is expressive enough to capture most computations supported by current data-centric distributed processing platforms. The main evidence of its effectiveness comes from its use in MRQL, which can run complex data analysis queries on a variety of distributed platforms, such as Map-Reduce, Spark, Flink, and Hama. Monoid homomorphisms capture associativity directly, which is essential for data parallelism. Monoid comprehensions extend list comprehension with heterogeneous collections, and group-by and order-by syntax, without the need of incorporating additional operations into the algebra. Our current work on incremental computing gives further evidence of the effectiveness of monoid homomorphisms, by giving a simple solution to the incrementalization problem by transforming queries to homomorphisms.

Acknowledgments: Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

References

U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An Experimental Analysis of Self-Adjusting Computation. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(1):3:1-53, 2009.

H. Aït-Kaci. An Abstract, Reusable, and Extensible Programming Language Design Architecture. In *In Search of Elegance in the Theory and Practice of Computation*, LNCS 8000, pp 112–166, 2013. Available at `http://hassan-ait-kaci.net/pdf/hak-opb.pdf`.

A. Alexandrov, A. Katsifodimos, G. Krastev, and V. Markl. Implicit Parallelism through Deep Language Embedding. In *SIGMOD Record*, 45(1):51–58, 2016.

M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational Data Processing in Spark. In *International Conference on Management of Data (SIGMOD)*, pp 1383–1394, 2015.

Apache Flink. `http://flink.apache.org/`. 2017.

Apache Giraph. `http://giraph.apache.org/`. 2017.

Apache Hadoop. `http://hadoop.apache.org/`. 2017.

Apache Hama. `http://hama.apache.org/`. 2017.

Apache Hive. `http://hive.apache.org/`. 2017.

Apache MRQL (incubating). `http://mrql.incubator.apache.org/`. 2017. The MRQL syntax is described at `http://wiki.apache.org/mrql/LanguageDescription`.

Apache Spark. `http://spark.apache.org/`. 2017.

F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. FAD, a Powerful and Simple Database Language. In *Proceedings of the International Conference on Very Large Data Bases*, pp 97–105, 1987.

D. Battre, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A Programming Model and Execution Framework for Web-Scale Analytical Processing. In *1st ACM Symposium on Cloud computing (SOCC'10)*, pp 119–130, 2010.

G. Blelloch. NESL: A Nested Data-Parallel Language. Technical report, Carnegie Mellon University, 1993. CMU-CS-93-129.

G. Blelloch and G. Sabot. Compiling Collection-Oriented Languages onto Massively Parallel Computers. In *Journal of Parallel and Distributed Computing*, 8(2):119–134, 1990.

O. Boykin, S. Ritchie, I. O'Connell, and J. Lin. Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. In *Proceedings of the VLDB Endowment (PVLDB)*, 7(13):1441–1451, 2014.

R. E. Bryant. Data-Intensive Scalable Computing for Scientific Applications. In *Computing in Science & Engineering*, 13(6):25–33, 2011.

P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. In *SIGMOD Record*, 23(11):87–96, 1994.

P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. In *Theoretical Computer Science*, 149(1):3–48, 1995.

R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1265–1276, 2008.

D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining (SDM)*, pp 442–446, 2004.

J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, 2004.

L. Fegaras. Supporting Bulk Synchronous Parallelism in Map-Reduce Queries. In *International Workshop on Data Intensive Computing in the Clouds (DataCloud)*, 2012.

L. Fegaras. Incremental Query Processing on Big Data Streams. In *IEEE Transactions on Knowledge and Data Engineering*, 28(11):2998–3012, 2016. Available at `https://lambda.uta.edu/tkde16-preprint.pdf`.

L. Fegaras. A Query Processing Framework for Array-Based Computations. In *International Conference on Database and Expert Systems Applications (DEXA)*, pp 240–254, 2016.

L. Fegaras, C. Li, U. Gupta, and J. J. Philip. XML Query Optimization in Map-Reduce. In *International Workshop on the Web and Databases (WebDB)*, 2011.

L. Fegaras, C. Li, and U. Gupta. An Optimization Framework for Map-Reduce Queries. In *International Conference on Extending Database Technology (EDBT)*, pp 26–37, 2012.

L. Fegaras and D. Maier. Towards an Effective Calculus for Object Query Languages. In *International Conference on Management of Data (SIGMOD)*, pp 47–58, 1995.

L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. In *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000. Available at `https://lambda.uta.edu/tods00.pdf`.

A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on top of Map-Reduce: the Pig Experience. In *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1414-1425, 2009.

J. Gibbons. Comprehending Ringads: For Phil Wadler on the Occasion of his 60th Birthday. In *A List of Successes That Can Change the World*. LNCS 9600, pp 132-151, 2016.

G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing Back Monad Comprehensions. In *Haskell Symposium*, pp 13–22, 2011.

M. Grabowski, J. Hidders, and J. Sroka. Representing MapReduce Optimisations in the Nested Relational Calculus. In *British National Conference on Big Data (BNCOD)*, pp 175–188, 2013.

T. Grust and M. H. Scholl. How to Comprehend Queries Functionally. In *Journal of Intelligent Information Systems*, 12(2-3):191–218, 1999.

T. Grust. Monad Comprehensions: A Versatile Representation for Queries. In *The Functional Approach to Data Management*, pp 288–311, Springer-Verlag, 2004

J. Holsch, M. Grossniklaus, and M. H. Scholl. Optimization of Nested Queries using the NF^2 Algebra. In *ACM SIGMOD International Conference on Management of Data*, pp 1765–1780, 2016.

M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *ACM SIGMOD International Conference on Management of Data*, pp 987–994, 2009.

J. Lin and C. Dyer. Data-Intensive Text Processing with MapReduce. Morgan & Claypool Publishers, 2010.

Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. In *Proceedings of the VLDB Endowment (PVLDB)*, 5(8):716–727, 2012.

G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a System for Large-Scale Graph Processing. In *ACM SIGMOD International Conference on Management of Data*, pp 135–146, 2010.

K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *ACM SIGPLAN PLDI*, pp 146–155, 2007.

C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-Foreign Language for Data Processing. In *ACM SIGMOD International Conference on Management of Data*, pp 1099–1110, 2008.

R. Power and J. Li. Piccolo: Building Fast, Distributed Programs with Partitioned Tables. In *Symposium on Operating System Design and Implementation (OSDI)*, 2010.

A. Shinnar, D. Cunningham, B. Herta, and V. Saraswat. M3R: Increased Performance for In-Memory Hadoop Jobs. In *Proceedings of the VLDB Endowment (PVLDB)*, 5(12):1736–1747, 2012.

G. L. Steele, Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *ICFP*, pp 1–2, 2009.

V. B.-Tannen, P. Buneman, and S. Naqvi. Structural Recursion as a Query Language. In *International Workshop on Database Programming Languages: Bulk Types and Persistent Data (DBPL)*, pp 9–19, 1991.

A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Antony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a Warehousing Solution over a Map-Reduce Framework. In *Proceedings of the VLDB Endowment (PVLDB)*, 2(2):1626–1629, 2009.

A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive: A Petabyte Scale Data Warehouse Using Hadoop. In *IEEE International Conference on Data Engineering (ICDE)*, pp 996–1005, 2010.

P. W. Trinder. Comprehensions, a Query Notation for DBPLs. In *International Workshop on Database Programming Languages (DBPL)*, pp 55–68, 1991.

P. Trinder and P. Wadler. Improving list comprehension database queries. In *TENCON*, pp 186–192, 1989.

L. G. Valiant. A Bridging Model for Parallel Computation. In *Communications of the ACM (CACM)*, 33(8):103–111, 1990.

P. Wadler. Comprehending Monads. In *ACM Symposium on Lisp and Functional Programming*, pp 61–78, 1990.

P. Wadler. List Comprehensions. Chapter 7 in *The Implementation of Functional Programming Languages*, by S. Peyton Jones, Prentice Hall, 1987.

P. Wadler and S. Peyton Jones. Comprehensive Comprehensions (Comprehensions with 'Order by' and 'Group by'). In *Haskell Symposium*, pp 61–72, 2007.

L. Wong. Kleisli, a Functional Query System. In *Journal of Functional Programming*, 10(1):19–56, 2000.

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

Appendix A: Proofs

Theorem A.1 (cMap Fusion)**.** *For all f, g, and S:*

$$\mathrm{cMap}(f, \mathrm{cMap}(g, S)) = \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), S) \tag{6}$$

Proof. We prove this theorem from the cMap definition using structural induction. Let S be a collection of type $\mathbb{T}_{\oplus}(\alpha)$, for some collection monoid $\oplus$. The theorem is true for $S = \mathbf{1}_{\oplus}$ and for $S = \mathbb{U}_{\oplus}(x)$, for all x. Assuming that it is true for $S = X$ and $S = Y$ (induction hypothesis), we prove that it is also true for $S = X \oplus Y$:

$$\begin{aligned}
& \mathrm{cMap}(f, \mathrm{cMap}(g, X \oplus Y)) \\
= \quad & [\text{ from Eq. (5), for some collection monoid } \otimes \text{ }] \\
& \mathrm{cMap}(f, \mathrm{cMap}(g, X) \otimes \mathrm{cMap}(g, Y)) \\
= \quad & [\text{ from Eq. (5), for some collection monoid } \odot \text{ }] \\
& \mathrm{cMap}(f, \mathrm{cMap}(g, X)) \odot \mathrm{cMap}(f, \mathrm{cMap}(g, Y)) \\
= \quad & [\text{ from induction hypothesis }] \\
& \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), X) \odot \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), Y) \\
= \quad & [\text{ from Eq. (5) }] \\
& \mathrm{cMap}(\lambda x.\, \mathrm{cMap}(f, g(x)), X \oplus Y)
\end{aligned}$$

□

Theorem A.2 (groupBy Unnesting)**.** *Unnesting a groupBy over a bag X returns the input bag X :*

$$\{\, (k, v) \mid (k, s) \in \mathrm{groupBy}(X),\, v \in s \,\} = X$$

Proof. We prove this theorem using structural induction. The theorem is true for an empty bag and for a singleton bag X. The inductive step

$$\{\, (k, v) \mid (k, s) \in \mathrm{groupBy}(X \uplus Y),\, v \in s \,\} = X \uplus Y$$

can be proven as follows. Let $X' = \mathrm{groupBy}(X)$ and $Y' = \mathrm{groupBy}(Y)$. Then, for

$$\begin{aligned}
M(X', Y') = & \{\, (k, a) \mid (k, a) \in X',\, (k', b) \in Y',\, k = k' \,\} \\
& \uplus \{\, (k, a) \mid (k, a) \in X',\, \forall (k', b) \in Y' : k' \neq k \,\}
\end{aligned} \tag{A 1}$$

we have

$$M(X', Y') = X' \tag{A 2}$$

because for each group-by key k, there is at most one $(k,a) \in X'$ and at most one $(k,b) \in Y'$. Then, we have:

$$
\begin{aligned}
& \{\, (k,v) \mid (k,s) \in \mathrm{groupBy}(X \uplus Y),\ v \in s \,\} \\
= \quad & [\text{ from the groupBy def. }] \\
& \{\, (k,v) \mid (k,s) \in (X' \Updownarrow_{\uplus} Y'),\ v \in s \,\} \\
= \quad & [\text{ from Eq. (8) }] \\
& \{\, (k,v) \mid (k,s) \in (\{\, (k, a \uplus b) \mid (k,a) \in X',\ (k',b) \in Y',\ k = k' \,\} \\
& \qquad\qquad\qquad\qquad \uplus \{\, (k,a) \mid (k,a) \in X',\ \forall (k',b) \in Y' : k' \neq k \,\} \\
& \qquad\qquad\qquad\qquad \uplus \{\, (k,b) \mid (k,b) \in Y',\ \forall (k',b) \in X' : k' \neq k \,\}),\ v \in s \,\} \\
= \quad & [\text{ from Eq. (23) }] \\
& \{\, (k,v) \mid (k,a) \in X',\ (k',b) \in Y',\ k = k',\ v \in (a \uplus b) \,\} \\
& \quad \uplus \{\, (k,v) \mid (k,a) \in X',\ \forall (k',b) \in Y' : k' \neq k, v \in a \,\} \\
& \quad \uplus \{\, (k,v) \mid (k,b) \in Y',\ \forall (k',b) \in X' : k' \neq k, v \in b \,\} \\
= \quad & [\text{ by expanding the qualifier } v \in (a \uplus b) \text{]} \\
& \{\, (k,v) \mid (k,a) \in X',\ (k',b) \in Y',\ k = k',\ v \in a \,\} \\
& \quad \uplus \{\, (k,v) \mid (k,a) \in X',\ (k',b) \in Y',\ k = k',\ v \in b \,\} \\
& \quad \uplus \{\, (k,v) \mid (k,a) \in X',\ \forall (k',b) \in Y' : k' \neq k, v \in a \,\} \\
& \quad \uplus \{\, (k,v) \mid (k,b) \in Y',\ \forall (k',b) \in X' : k' \neq k, v \in b \,\} \\
= \quad & [\text{ from Eq. (A 1) }] \\
& \{\, (k,v) \mid (k,a) \in M(X',Y'),\ v \in a \,\} \uplus \{\, (k,v) \mid (k,b) \in M(Y',X'),\ v \in b \,\} \\
= \quad & [\text{ from Eq. (A 2) }] \\
& \{\, (k,v) \mid (k,a) \in X',\ v \in a \,\} \uplus \{\, (k,v) \mid (k,b) \in Y',\ v \in b \,\} \\
= \quad & [\text{ from induction hypothesis }] \\
& X \uplus Y
\end{aligned}
$$

□

Theorem A.3 (groupBy Fusion)**.** *For any collection X of type $\mathbb{T}_{\oplus}(\kappa \times \alpha)$:*

$$\mathrm{groupBy}(\mathrm{groupBy}(X)) \;=\; \mathrm{cMap}(\lambda(k,s).\, \{\!\{(k, \{\!\{s\}\!\})\}\!\}, \mathrm{groupBy}(X)) \tag{10}$$

Proof. We prove this theorem using structural induction. The theorem is true for an empty and for a singleton collection X. The inductive step for $X = X_1 \oplus X_2$ can be proven as follows:

$$
\begin{aligned}
& \mathrm{groupBy}(\mathrm{groupBy}(X)) \;=\; \mathrm{groupBy}(\mathrm{groupBy}(X_1 \oplus X_2)) \\
= \quad & [\text{ from groupBy def. }] \\
& \mathrm{groupBy}(\mathrm{groupBy}(X_1) \Updownarrow_{\oplus} \mathrm{groupBy}(X_2))
\end{aligned}
$$

$=$ [from groupBy def.]

$\quad \text{groupBy}(\text{groupBy}(X_1)) \Updownarrow_{\Updownarrow_\oplus} \text{groupBy}(\text{groupBy}(X_2))$

$=$ [from induction hypothesis]

$\quad \text{cMap}(\lambda(k,s).\, \{\!\{(k, \{\!\{s\}\!\})\}\!\}, \text{groupBy}(X_1)) \Updownarrow_{\Updownarrow_\oplus} \text{cMap}(\lambda(k,s).\, \{\!\{(k, \{\!\{s\}\!\})\}\!\}, \text{groupBy}(X_2))$

$=$ [from $\Updownarrow$ def.]

$\quad \{\, (k, \{\!\{a\}\!\} \Updownarrow_\oplus \{\!\{b\}\!\}) \,|\, (k,a) \in \text{groupBy}(X_1),\, (k,b) \in \text{groupBy}(X_2) \,\} \uplus \cdots$

$=$ [from groupBy def.]

$\quad \text{cMap}(\lambda(k,s).\, \{\!\{(k, \{\!\{s\}\!\})\}\!\}, \text{groupBy}(X_1 \oplus X_2))$

$= \text{cMap}(\lambda(k,s).\, \{\!\{(k, \{\!\{s\}\!\})\}\!\}, \text{groupBy}(X))$

where the $\cdots$ are terms that have been omitted for brevity. □

Theorem A.4 ($\Uparrow_\oplus$ is a Monoid)**.** *The operation $\Uparrow_\oplus$, defined in Eqs. (13a)-(13c), is a monoid in the range of $\Uparrow_\oplus$.*

Proof. From Eq. (13c), $[\,] \Uparrow_\oplus Y = Y$. We prove $X \Uparrow_\oplus [\,] = X$ using induction on X. It is true for $X = [\,]$ and $X = [(k,v)]$. For X, X_1, X_2 in the range of $\Uparrow_\oplus$, we have $X = X_1 +\!\!+ X_2 = X_1 \Uparrow_\oplus X_2$. Assuming $X \Uparrow_\oplus [\,] = X$ is true for X_1 and X_2, we have

$\quad X \Uparrow_\oplus [\,] \;=\; (X_1 +\!\!+ X_2) \Uparrow_\oplus [\,]$

$=$ [from Eq. (13a)]

$\quad X_1 \Uparrow_\oplus (X_2 \Uparrow_\oplus [\,])$

$=$ [from induction hypothesis]

$\quad X_1 \Uparrow_\oplus X_2 \;=\; X$

If $X = X_1 +\!\!+ X_2 = X_1 \Uparrow_\oplus X_2$, then, from Eq. (13a), $(X_1 +\!\!+ X_2) \Uparrow_\oplus Y = X_1 \Uparrow_\oplus (X_2 \Uparrow_\oplus Y)$, which implies the associativity law $(X_1 \Uparrow_\oplus X_2) \Uparrow_\oplus Y = X_1 \Uparrow_\oplus (X_2 \Uparrow_\oplus Y)$. □

Theorem A.5. *The group-by and order-by elimination rules in Eqs. (19a) and (19b) are confluent, that is, if there are multiple group-by and order-by qualifiers in a monoid comprehension, the order that these qualifiers are eliminated using these rules is insignificant.*

Proof. We prove this theorem for two group-bys in the same comprehension:

$\quad \{\, e \,|\, \overline{q_1}, \textbf{group by}\, p_1 : e_1, \overline{q_2}, \textbf{group by}\, p_2 : e_2, \overline{q_3} \,\}$

$=$ [from Eq. (19a) applied to the first **group by**]

$\quad \{\, e \,|\, (p_1, s_1) \in \text{groupBy}(\{\, (e_1, \mathscr{V}^{p_1}_{\overline{q_1}}) \,|\, \overline{q_1} \,\}),$

$\qquad \forall v \in \mathscr{V}^{p_1}_{\overline{q_1}} :\; \textbf{let}\; v = \{\, v \,|\, \mathscr{V}^{p_1}_{\overline{q_1}} \in s_1 \,\}, \overline{q_2},$

$\qquad \textbf{group by}\, p_2 : e_2, \overline{q_3} \,\}$

$$
\begin{aligned}
= \quad & [\text{ from Eq. (19a) with } \overline{q} = \overline{q_1}, \textbf{group by}\, p_1 : e_1, \overline{q_2} \,] \\
& \{\, e \mid (p_2, s_2) \in \text{groupBy}(\{\, (e_2, \mathcal{V}^{p_2}_{\overline{q_2}}) \mid (p_1, s_1) \in \text{groupBy}(\{\, (e_1, \mathcal{V}^{p_1}_{\overline{q_1}}) \mid \overline{q_1} \,\}), \\
& \qquad\qquad\qquad\qquad\qquad\qquad \forall v \in \mathcal{V}^{p_1}_{\overline{q_1}} :\ \textbf{let}\, v = \{\, v \mid \mathcal{V}^{p_1}_{\overline{q_1}} \in s_1 \,\}, \overline{q_2} \,\}), \\
& \qquad \forall v \in \mathcal{V}^{p_2}_{\overline{q_2}} :\ \textbf{let}\, v = \{\, v \mid \mathcal{V}^{p_2}_{\overline{q_2}} \in s_2 \,\}, \overline{q_3} \,\} \\
= \quad & [\text{ from Eq. (19a) applied to the inner comprehension }] \\
& \{\, e \mid (p_2, s_2) \in \text{groupBy}(\{\, (e_2, \mathcal{V}^{p_2}_{\overline{q_2}}) \mid \overline{q_1}, \textbf{group by}\, p_1 : e_1, \overline{q_2} \,\}, \\
& \qquad \forall v \in \mathcal{V}^{p_2}_{\overline{q_2}} :\ \textbf{let}\, v = \{\, v \mid \mathcal{V}^{p_2}_{\overline{q_2}} \in s_2 \,\}, \overline{q_3} \,\}
\end{aligned}
$$

which is equal to the original term with the second group-by translated first. □

Theorem A.6 (Comprehension Unnesting)**.** *Monoid comprehensions without group-by or order-by qualifiers satisfy:*

$$\{\, e \mid \overline{q_1}, p \in \{\, e' \mid \overline{q_3} \,\}, \overline{q_2} \,\} = \{\, e \mid \overline{q_1}, \overline{q_3}, \textbf{\textit{let}}\; p = e', \overline{q_2} \,\} \tag{23}$$

for all qualifiers $\overline{q_1}$, $\overline{q_2}$, and $\overline{q_3}$, for any pattern p, and for all expressions e and e'.

Proof. It is sufficient to prove:

$$\{\, e \mid p \in \{\, e' \mid \overline{q_3} \,\}, \overline{q_2} \,\} = \{\, e \mid \overline{q_3}, \textbf{let}\; p = e', \overline{q_2} \,\}$$

since both comprehensions in Eq. (23) start with the same qualifiers $\overline{q_1}$. We prove this equation using induction over the first qualifier of $\overline{q_3}$, which we assume is a generator qualifier (the proof for the other qualifiers types is easier). That is, we assume the law is true for $\overline{q'_3}$ and we prove it for $\overline{q_3} = p_3 \in e_3, \overline{q'_3}$ (comprehension subscripts have been omitted):

$$
\begin{aligned}
& \{\, e \mid p \in \{\, e' \mid \overline{q_3} \,\}, \overline{q_2} \,\} \;=\; \{\, e \mid p \in \{\, e' \mid p_3 \in e_3, \overline{q'_3} \,\}, \overline{q_2} \,\} \\
= \quad & [\text{ from Rule (22b) }] \\
& \{\, e \mid p \in \text{cMap}(\lambda p_3.\{\, e' \mid \overline{q'_3} \,\}, e_3), \overline{q_2} \,\} \\
= \quad & [\text{ from Rule (22b) }] \\
& \text{cMap}(\lambda p.\{\, e \mid \overline{q_2} \,\}, \text{cMap}(\lambda p_3.\{\, e' \mid \overline{q'_3} \,\}, e_3)) \\
= \quad & [\text{ from Eq. (6) }] \\
& \text{cMap}(\lambda p_3.\text{cMap}(\lambda p.\{\, e \mid \overline{q_2} \,\}, \{\, e' \mid \overline{q'_3} \,\}), e_3) \\
= \quad & [\text{ from Rule (22b) }] \\
& \text{cMap}(\lambda p_3.\{\, e \mid p \in \{\, e' \mid \overline{q'_3} \,\}, \overline{q_2} \,\}, e_3) \\
= \quad & [\text{ from Rule (22b) }] \\
& \{\, e \mid p_3 \in e_3, p \in \{\, e' \mid \overline{q'_3} \,\}, \overline{q_2} \,\} \\
= \quad & [\text{ from induction hypothesis }] \\
& \{\, e \mid p_3 \in e_3, \overline{q'_3}, \textbf{let}\; p = e', \overline{q_2} \,\} \;=\; \{\, e \mid \overline{q_3}, \textbf{let}\; p = e', \overline{q_2} \,\}
\end{aligned}
$$

□

Theorem A.7 (Nested cMaps to a coGroup)**.** *Let $F(X,Y)$ be a nested cMap over bags*

$$\begin{aligned} F(X,Y) &= \text{cMap}(\lambda x.\, g(\text{cMap}(\lambda y.\, h(x,y),\, Y)), X) \\ &= \{\, z \mid x \in X,\, z \in g(\{\, w \mid y \in Y,\, w \in h(x,y) \,\}) \,\} \end{aligned}$$

for some term functions g and h. If there are term functions k_1 and k_2 such that $h(x,y) = \textbf{\textit{if}}\ (k_1(x) = k_2(y))\ \textbf{\textit{then}}\ h(x,y)\ \textbf{\textit{else}}\ \{\!\{\,\}\!\}$, then $F(X,Y)$ is equal to:

$$\begin{aligned} &\text{cMap}(\lambda(k,(xs,ys)).\, F(xs,ys), \\ &\qquad \text{coGroup}(\text{cMap}(\lambda x.\, \{\!\{(k_1(x),x)\}\!\}, X),\, \text{cMap}(\lambda y.\, \{\!\{(k_2(y),y)\}\!\}, Y))) \end{aligned}$$

Proof. It can be proven (with a proof similar to that of Eq. (9)) that, if a coGroup is over bags, then each of the coGroup inputs can be derived from the coGroup result:

$$\{\, (k,x) \mid (k,(s_1,s_2)) \in \text{coGroup}(X,Y),\, x \in s_1 \,\} \;=\; X \tag{A 3}$$

$$\{\, (k,y) \mid (k,(s_1,s_2)) \in \text{coGroup}(X,Y),\, y \in s_2 \,\} \;=\; Y \tag{A 4}$$

The theorem precondition is equivalent to $h(x,y) = \{\, z \mid k_1(x) = k_2(y),\, z \in h(x,y) \,\}$. For $X' = \text{cMap}(\lambda x.\, \{\!\{(k_1(x),x)\}\!\}, X)$ and $Y' = \text{cMap}(\lambda y.\, \{\!\{(k_2(y),y)\}\!\}, Y)$, we have:

$$\begin{aligned} &F(X,Y) = \{\, z \mid x \in X,\, z \in g(\{\, w \mid y \in Y,\, w \in h(x,y) \,\}) \,\} \\ = \quad &[\text{ from the precondition }] \\ &\{\, z \mid x \in X,\, z \in g(\{\, w \mid y \in Y,\, w \in \{\, z \mid k_1(x) = k_2(y),\, z \in h(x,y) \,\} \,\}) \,\} \\ = \quad &[\text{ from Eq. (23) }] \\ &\{\, z \mid x \in X,\, z \in g(\{\, w \mid y \in Y,\, k_1(x) = k_2(y),\, w \in h(x,y) \,\}) \,\} \\ = \quad &[\text{ from the } X' \text{ and } Y' \text{ def. }] \\ &\{\, z \mid (k,x) \in X',\, z \in g(\{\, w \mid (k',y) \in Y',\, k = k',\, w \in h(x,y) \,\}) \,\} \\ = \quad &[\text{ from Eq. (A 3) }] \\ &\{\, z \mid (k,(xs,ys)) \in \text{coGroup}(X',Y'),\, x \in xs, \\ &\qquad z \in g(\{\, w \mid (k',y) \in Y',\, k = k',\, w \in h(x,y) \,\}) \,\} \\ = \quad &[\text{ from Eq. (A 4) }] \\ &\{\, z \mid (k,(xs,ys)) \in \text{coGroup}(X',Y'),\, x \in xs, \\ &\qquad z \in g(\{\, w \mid (k',(xs',ys')) \in \text{coGroup}(X',Y'),\, y \in ys',\, k = k',\, w \in h(x,y) \,\}) \,\} \end{aligned}$$

$= \quad$ [the qualifier $(k',(xs',ys')) \in \text{coGroup}(X',Y')$ can be removed because it is over the same key $k'=k$ as in $(k,(xs,ys)) \in \text{coGroup}(X',Y')$]

$$\{\, z \mid (k,(xs,ys)) \in \text{coGroup}(X',Y'),\, x \in xs,\, z \in g(\{\, w \mid y \in ys,\, w \in h(x,y)\,\})\,\}$$

$= \quad$ [from Eq. (23)]

$$\{\, c \mid (k,(xs,ys)) \in \text{coGroup}(X',Y'),\, c \in \{\, z \mid x \in xs,\, z \in g(\{\, w \mid y \in ys,\, w \in h(x,y)\,\})\,\}\,\}$$

$= \quad$ [from the F def.]

$$\{\, c \mid (k,(xs,ys)) \in \text{coGroup}(X',Y'),\, c \in F(xs,ys)\,\}$$

$= \quad$ [from Eqs. (22b)-(22e)]

$$\text{cMap}(\lambda(k,(xs,ys)).\, F(xs,ys),\, \text{coGroup}(X',Y')) \qquad \square$$

Theorem A.8 (coGroup-groupBy Fusion). *For all bags X and Y:*

$$\begin{aligned}&\text{coGroup}(\text{cMap}(\lambda(k,s).\,\{\!\{(k,f(s))\}\!\},\, \text{groupBy}(X)),Y)\\ &\quad = \text{cMap}(\lambda(k,(xs,ys)).\,\{\!\{(k,(\{\!\{f(xs)\}\!\},ys))\}\!\},\, \text{coGroup}(X,Y))\end{aligned}$$

Proof. First, for all bags X and Y, we have:

$$\text{coGroup}(X,Y)$$

$= \quad$ [from Eq. (16)]

$$\{\, (k,(xs,ys)) \mid (k,(xs,ys)) \in (\text{coGroup}(X,\{\!\{\,\}\!\}) \Updownarrow_{\uplus\star\uplus} \text{coGroup}(\{\!\{\,\}\!\},Y))\,\}$$

$= \quad$ [from Eqs. (17a) and (17b)]

$$\begin{aligned}\{\, (k,(xs,ys)) \mid (k,(xs,ys)) \in (&\{\, (k,(xs,\{\!\{\,\}\!\})) \mid (k,xs) \in \text{groupBy}(X)\,\}\\ &\Updownarrow_{\uplus\star\uplus} \{\, (k,(\{\!\{\,\}\!\},ys)) \mid (k,ys) \in \text{groupBy}(Y)\,\})\,\}\end{aligned}$$

$= \quad$ [from Eq. (8)]

$$\begin{aligned}\{\, (k,(xs,ys)) \mid (k,(xs,ys)) \in (&\{\, (k,(xs,ys)) \mid (k,xs) \in \text{groupBy}(X),\\ &(k',ys) \in \text{groupBy}(Y),\, k=k'\,\} \uplus \cdots)\,\}\end{aligned}$$

$$= \{\, (k,(xs,ys)) \mid (k,xs) \in \text{groupBy}(X),\, (k',ys) \in \text{groupBy}(Y),\, k=k'\,\} \uplus \cdots$$

Using this equation, we have:

$$\text{cMap}(\lambda(k,(xs,ys)).\,\{\!\{(k,(\{\!\{f(xs)\}\!\},ys))\}\!\},\, \text{coGroup}(X,Y))$$

$$= \{\, (k,(\{\!\{f(xs)\}\!\},ys)) \mid (k,(xs,ys)) \in \text{coGroup}(X,Y)\,\}$$

$= \quad$ [from the previous equation for coGroup]

$$\{\, (k,(\{\!\{f(xs)\}\!\},ys)) \mid (k,xs) \in \text{groupBy}(X),\, (k',ys) \in \text{groupBy}(Y),\, k=k'\,\} \uplus \cdots$$

$= \quad$ [from Eq. (10)]

$$\{\, (k,(f(xs),ys)) \mid (k,xs) \in \text{groupBy}(\text{groupBy}(X)),\, (k',ys) \in \text{groupBy}(Y),\, k=k'\,\} \uplus \cdots$$

$= \quad$ [from the previous equation for coGroup]

$$\text{coGroup}(\text{cMap}(\lambda(k,s).\,\{\!\{(k,f(s))\}\!\},\, \text{groupBy}(X)),Y)$$

where the $\cdots$ are terms that have been omitted for brevity. $\square$

Theorem A.9 (coGroup to groupBy)**.** *For all bags X and Y:*

$$\begin{array}{rcl}\mathrm{coGroup}(X,Y) & = & \mathrm{cMap}(\lambda(k,s).\,\{\!\{(k,(l'(s),r'(s)))\}\!\},\\ & & \quad \mathrm{groupBy}(l(X)\uplus r(Y)))\end{array}$$

where

$$\begin{array}{rcl}l(X) & = & \mathrm{cMap}(\lambda(k,a).\,\{\!\{(k,\boldsymbol{inL}(a))\}\!\},X)\\ r(Y) & = & \mathrm{cMap}(\lambda(k,b).\,\{\!\{(k,\boldsymbol{inR}(b))\}\!\},Y)\\ l'(s) & = & \mathrm{cMap}(\lambda v.\ \boldsymbol{case}\ v\ \boldsymbol{of}\ \boldsymbol{inL}(a)\Rightarrow\{\!\{a\}\!\}\mid\boldsymbol{inR}(b)\Rightarrow\{\!\{\,\}\!\},s)\\ r'(s) & = & \mathrm{cMap}(\lambda v.\ \boldsymbol{case}\ v\ \boldsymbol{of}\ \boldsymbol{inL}(a)\Rightarrow\{\!\{\,\}\!\}\mid\boldsymbol{inR}(b)\Rightarrow\{\!\{b\}\!\},s)\end{array}$$

Proof. We use the following two equations. which can be proven by induction on bags X and Y:

$$\begin{array}{rcl}\mathrm{groupBy}(X) & = & \{\,(k,l'(s))\mid(k,s)\in\mathrm{groupBy}(l(X))\,\}\\ \mathrm{groupBy}(Y) & = & \{\,(k,r'(s))\mid(k,s)\in\mathrm{groupBy}(r(Y))\,\}\end{array}$$

Based on these equations, we have:

$$\begin{array}{ll} & \mathrm{coGroup}(X,Y)\\ = & [\text{ from the proof of Theorem A.8 }]\\ & \{\,(k,(xs,ys))\mid(k,xs)\in\mathrm{groupBy}(X),\,(k',ys)\in\mathrm{groupBy}(Y),\,k=k'\,\}\uplus\cdots\\ = & [\text{ from the previous equations }]\\ & \{\,(k,(xs,ys))\mid(k,xs)\in\{\,(k,l'(s))\mid(k,s)\in\mathrm{groupBy}(l(X))\,\},\\ & \qquad\qquad (k',ys)\in\{\,(k,r'(s))\mid(k,s)\in\mathrm{groupBy}(r(Y))\,\},\,k=k'\,\}\uplus\cdots\\ = & [\text{ from Eq. (23) }]\\ & \{\,(k,(l'(s_1),r'(s_2)))\mid(k,s_1)\in\mathrm{groupBy}(l(X)),\\ & \qquad\qquad\qquad (k',s_2)\in\mathrm{groupBy}(r(Y)),\,k=k'\,\}\uplus\cdots\\ = & [\text{ from Eq. (8) }]\\ & \{\,(k,(l'(s),r'(s)))\mid(k,s)\in(\mathrm{groupBy}(l(X))\Updownarrow_{\uplus}\mathrm{groupBy}(r(Y)))\,\}\\ = & [\text{ from the groupBy def. }]\\ & \{\,(k,(l'(s),r'(s)))\mid(k,s)\in\mathrm{groupBy}(l(X)\uplus r(Y))\,\}\end{array}$$

where the $\cdots$ are terms that have been omitted for brevity. □