Compile-Time Code Generation for Embedded Data-Intensive Query Languages

Leonidas Fegaras, Md Hasanuzzaman Noor

University of Texas at Arlington, CSE
Arlington, TX 76019
fegaras@cse.uta.edu, mdhasanuzzaman.noor@mavs.uta.edu

***Abstract*—Many emerging Big Data programming environments, such as Spark and Flink, provide powerful APIs that are inspired by functional programming. However, because of the complexity involved in developing and fine-tuning data analysis applications using the provided APIs, many programmers prefer to use declarative languages, such as Hive and Spark SQL, to code their distributed applications. Unfortunately, current data analysis query languages, which are typically based on the relational model, cannot effectively capture the rich data types and computations required for complex data analysis applications. Furthermore, these query languages are not well-integrated with the host programming language, as they are based on an incompatible data model, and are checked for correctness at run-time, which results in a significantly longer program development time. To address these shortcomings, we introduce a new query language for data-intensive scalable computing, called DIQL, that is deeply embedded in Scala, and a query optimization framework that optimizes and translates DIQL queries to byte code at compile-time. In contrast to other query languages, our query embedding eliminates impedance mismatch as any Scala code can be seamlessly mixed with SQL-like syntax, without having to add any special declaration. DIQL supports nested collections and hierarchical data and allows query nesting at any place in a query. With DIQL, programmers can express complex data analysis tasks, such as PageRank and matrix factorization, using SQL-like syntax exclusively. The DIQL query optimizer can find any possible join in a query, including joins hidden across deeply nested queries, thus unnesting any form of query nesting. Currently, DIQL can run on three Big Data platforms: Apache Spark, Apache Flink, and Twitter's Cascading/Scalding.**

I. Introduction

A. Motivation

New frameworks in Big Data analytics have become indispensable tools for large-scale data mining and scientific discoveries. One of the earliest Big Data analysis frameworks was the Map-Reduce model, which was introduced by Google in 2004 [14] and later became popular as an open-source system with Apache Hadoop [5]. Soon it became clear that the Map-Reduce model has some important drawbacks that impose a high overhead to complex workflows and graph algorithms, such as, storing the intermediate results between consecutive Map-Reduce jobs in secondary storage. To address some of the shortcomings of the Map-Reduce model, new alternative frameworks have been introduced recently. Among them, the most promising frameworks that seem to be good alternatives to Map-Reduce while addressing its drawbacks are Apache Spark [7] and Apache Flink [4], which cache most of their distributed data in the memory of the worker nodes. We collectively refer to these data-intensive distributed computing environments as DISC (Data-Intensive Scalable Computing) programming environments.

Some of the emerging DISC programming environments 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. Many of these frameworks provide a Scala-based API, because Scala is emerging as the functional language of choice for Big Data analytics. Examples of such APIs include the Scala-based APIs for Hadoop Map-Reduce, Scalding [27] and Scrunch [28], and the Hadoop alternatives, Spark [7] and Flink [4]. These APIs are based on distributed collections that resemble regular Scala data collections as they support similar methods. Although these distributed collections come under different names, such as TypedPipes in Scalding, PCollections in Scrunch, RDDs in Spark, and DataSets in Flink, they all represent immutable homogeneous collections of data distributed across the compute nodes of a cluster and they are very similar. By providing an API that is similar to the Scala collection API, programmers already familiar with Scala programming can start developing distributed applications with minimal training. Furthermore, many Big Data analysis applications need to work on nested collections, because, unlike relational databases, they need to analyze data in their native format, as they become available, without having to normalize these data into flat relations first and then reconstruct the data during querying using expensive joins. Thus, data analysis applications often work on distributed collections that contain nested sub-collections. While outer collections need to be distributed to be processed in parallel, the inner sub-collections must be stored in memory and processed as regular Scala collections. By providing similar APIs for both distributed datasets and in-memory collections, these frameworks provide a uniform way for processing data collections that simplifies program development considerably.

Although DISC frameworks provide powerful APIs that are simple to understand, it is hard to develop non-trivial applications coded in a general-purpose programming language,

especially when the focus is in optimizing performance. Much of the time spent programming these APIs is for addressing the intricacies and avoiding the pitfalls inherent to these frameworks. For instance, if the functional argument of a Spark operation accesses a non-local variable, the value of this variable is implicitly serialized and broadcast to all the worker nodes that evaluate this function. This broadcasting is completely hidden from the programmers, who must now make sure that there is no accidental reference to a large data structure within the functional parameters. Furthermore, the implicit broadcasting of non-local variables is less efficient than the explicit peer-to-peer broadcast operation in Spark, which uses faster serialization formats. A common error made by novice Spark programmers is to try to operate on an RDD from within the functional argument of another RDD operation, only to discover at run-time that this is impossible since functional arguments are evaluated by each worker node while RDDs must be distributed across the worker nodes. Instead, programmers should either broadcast the inner RDD to the worker nodes before the outer operation or use a join to combine the two RDDs. More importantly, some optimizations in the core Spark API, such as column pruning and selection pushdown, must be done by hand, which is very hard and may result to obscure code that does not reflect the intended application logic. Very often, one may have to choose among alternative operations that have the same functionality but different performance characteristics, such as using a reduceByKey operation instead of a groupByKey followed by a reduce operation. Furthermore, the core Spark API does not provide alternative join algorithms, such as broadcast and sort-merge joins, thus leaving the development of these algorithms to the programmers, which duplicates efforts and may lead to suboptimal performance.

In addition to hand-optimizing programs expressed in these APIs, there are many configuration parameters to adjust for better performance that overwhelm non-expert users. To find an optimal configuration for a certain data analysis application in Spark, one must decide how many executors to use, how many cores and how much memory to allocate per executor, how many partitions to split the data, etc. Furthermore, to improve performance, one can specify the number of reducers in Spark operations that cause data shuffling, instead of using the default, or even repartition the data in some cases to modify the degree of parallelism or to reduce data skew. Such adjustments are unrelated to the application logic but affect performance considerably.

Because of the complexity involved in developing and fine-tuning data analysis applications using the provided APIs, most programmers prefer to use declarative domain-specific languages (DSLs), such as Hive [6], Pig [26], MRQL [17], and Spark SQL [8], to code their distributed applications, instead of coding them directly in an algorithmic language. Most of these DSL-based frameworks though provide a limited syntax for operating on data collections, in the form of simple joins and group-bys. Some of them have limited support for nested collections and hierarchical data, and cannot express complex data analysis tasks, such as PageRank and data clustering, using DSL syntax exclusively. Spark DataFrames, for example, allows nested collections but provides a naïve way to process them: one must use the 'explode' operation on a nested collection in a row to flatten the row to multiple rows. Hive supports 'lateral views' to avoid creating intermediate tables when exploding nested collections. Both Hive and DataFrames treat in-memory collections differently from distributed collections, resulting to an awkward way to query nested collections.

One of the advantages of using DSL-based systems in developing DISC applications is that these systems support automatic program optimization. Such program optimization is harder to achieve in an API-based system. Some API-based systems though have found ways to circumvent this shortcoming. The evaluation of RDD 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 of the 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 [30]. 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 DataFrames are translated and optimized to RDD workflows at run-time using the Catalyst architecture. The optimizations include pushing down predicates, column pruning, and constant folding, but there are also plans for providing cost-based query optimizations, such as join reordering. Spark SQL though cannot handle most forms of nested queries and does not support iteration, thus making it inappropriate for complex data analysis applications. One common characteristic of most DSL-based frameworks is that, after optimizing a DSL program, they compile it to machine code at run-time, using an optimizing code generator, such as LLVM, or run-time reflection in Java or Scala.

Our goal is to design a query language for DISC applications that can be fully and effectively embedded into a host programming language (PL). To minimize impedance mismatch, the query data model must be equivalent to that of the PL. This restriction alone makes relational query languages a poor choice for query embedding since they cannot embed nested collections from the host PL into a query. Furthermore, data-centric query languages must work on special collections, which may have different semantics from the collections provided by the host PL. For instance, DISC collections are distributed across the worker nodes. To minimize impedance mismatch, data-centric and PL collections must be indistinguishable in the query language, although they may be

processed differently by the query system. In addition, the presence of null values complicates query embedding because, unlike SQL, which uses 3-valued logic to handle null markers introduced by outer joins, most PLs do not provide a standardized way to treat nulls. Instead, one would have to write explicit code in the query to handle them. Thus, embedding PL-defined UDF calls and custom aggregations in a query can either be done by writing explicit code in the query to handle nulls before these calls or by avoiding null values in the query data model altogether. We adopt the latter approach because we believe that nulls complicate query semantics and query embedding, and may result to obscure queries. Banning null values though means banning outer joins, which are considered important for practical query languages. In SQL, for example, matrix addition of sparse matrices can be expressed as a full outer join, to handle the missing entries in sparse matrices. In a data model that supports nested collections though, outer joins can be effectively captured by a coGroup operation (as defined in Pig, Spark, and many other DISC frameworks), which does not introduce any nulls. This implies that we can avoid generating null values as long as we provide syntax in the query language that captures any coGroup operation. Furthermore, most outer semijoins, which are very common in complex data-centric applications, can be more naturally expressed as nested queries. Thus, a query language must allow any arbitrary form of query nesting, while the query processor must be able to evaluate nested queries efficiently using joins. Consequently, query unnesting (i.e., finding all possible joins across nested queries) is crucial for embedded query languages. Current DISC query languages have very limited support for query nesting. Hive and Spark SQL, for example, support very simple forms of query nesting in predicates, thus forcing the programmers to use explicit outer joins to simulate the other forms of nested queries and write code to handle nulls.

B. Our Approach

We present a new query language for DISC systems, called DIQL (the Data-Intensive Query Language), that is deeply embedded in Scala, and a query optimization framework that optimizes DIQL queries and translates them to Java byte code at compile-time. DIQL is designed to support multiple Scala-based APIs for distributed processing by abstracting their distributed data collections as a *DataBag*, which is a bag distributed across the worker nodes of a computer cluster. Currently, DIQL supports three Big Data platforms that provide different APIs and performance characteristics: Apache Spark, Apache Flink, and Twitter's Cascading/Scalding. Unlike other query languages for DISC systems, DIQL can uniformly work on both distributed and in-memory collections using the same syntax. DIQL allows seamless mixing of native Scala code, which may contain UDF calls, with SQL-like query syntax, thus combining the flexibility of general-purpose programming languages with the declarativeness of database query languages. DIQL queries may use any Scala pattern, may access any Scala variable, and may embed any Scala code without any marshaling. More importantly, DIQL queries can use the core Scala libraries and tools as well as user-defined classes without having to add any special declaration. This tight integration with Scala eliminates impedance mismatch, reduces program development time, and increases productivity, since it finds syntax and type errors at compile-time. DIQL supports nested collections and hierarchical data, and allows query nesting at any place in a query. The query optimizer can find any possible join, including joins hidden across deeply nested queries, thus unnesting any form of query nesting. The DIQL algebra, which is based on monoid homomorphisms, can capture all the language features using a very small set of homomorphic operations. Monoids and monoid homomorphisms fully capture the functionality provided by current DSLs for DISC processing by directly supporting operations, such as group-by, order-by, aggregation, and joins on complex collections.

As an example of a DIQL query evaluated on Spark, consider matrix multiplication. We can represent a sparse matrix M as a distributed collection of type RDD[(Double,Int,Int)] in Spark, so that a triple (v,i,j) in this collection represents the matrix element $v = M_{ij}$. Then the matrix multiplication between two sparse matrices X and Y can be expressed as follows in DIQL:

select $(+/z,\ i,\ j)$
from $(x,i,k) <- X,\ (y,k\_,j) <- Y,\ z = x * y$
where $k == k\_$ **group by** (i,j)

where X and Y are embedded Scala variables of type RDD[(Double,Int,Int)]. This query retrieves the values $X_{ik} \in X$ and $Y_{kj} \in Y$ for all i,j,k, and 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 a bag of 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 matrix indexes i and j, the variable z is lifted to a bag of numerical values $X_{ik} * Y_{kj}$, for all k. Hence, the aggregation $+/z$ will sum up all the values in the bag z, deriving $\sum_k X_{ik} * Y_{kj}$ for the ij element of the resulting matrix.

As another example, consider the matrix addition between the two sparse matrices X and Y. Given that missing values in sparse matrices correspond to zero values, matrix addition should be implemented as a full outer join, not an inner join. This operation is specified as follows in DIQL:

select $(+/(x{+\!\!+}y),\ i,\ j)$
from $(x,i,j) <- X$ **group by** (i,j)
from $(y,i\_,j\_) <- Y$ **group by** $(i\_,j\_)$

Each of the two group-by clauses in this query groups only the variables in their associated from-clause: X values are grouped by (i,j) and Y values are grouped by $(i\_,j\_)$. Furthermore, the two group keys are set to be equal, $(i,j) = (i\_,j\_)$. The first (resp. second) group-by lifts the variable x (resp. y) to a bag of Double, which may be either empty or singleton. That is, the concatenation $x{+\!\!+}y$ may contain one or two

values, which are added together in $+/(x{+}{+}y)$. In contrast to a relational outer join, this query does not introduce any null values since it is equivalent to a coGroup operation.

The intended users of DIQL are developers of DISC applications who 1) have a preference in declarative query languages, such as SQL, over the higher-order functional programming style used by DISC APIs, 2) want to use a full fledged query language that is tightly integrated with the host language, combining the flexibility of general-purpose programming languages with the declarativeness of database query languages, 3) want to express their applications in a platform-independent language and experiment with multiple DISC platforms without modifying their programs, 4) want to focus on the programming logic without having to add obscure and platform-dependent performance details to the code, and 5) want to achieve good performance by relying on a sophisticated query optimizer.

Currently, the DIQL query optimizer is not cost-based; instead, it uses a special syntactic hint in a query to guide the optimizer. More specifically, instead of a qualifier $p<-e$ in the from-clause of a query, one may use the syntax $p<--e$ to indicate that the traversed collection e is small enough to fit in a worker's memory so that the optimizer may consider using a broadcast join to implement this traversal. But, as we know from traditional databases, cost-based optimizations are more effective at run-time, when there is statistical information available on the input and intermediate datasets. Other statically-typed query systems, such as DryadLINQ [33] and Emma [3], perform both static and dynamic optimizations; they perform static optimizations using greedy heuristics to generate a query graph, which is further optimized at run-time based on cost. Dynamic query optimization introduces a run-time overhead, which may become substantial if the query is embedded in a loop. More importantly, generating a static query representation to be staged for execution at run-time complicates query embedding. In DryadLINQ, for example, embedded values are serialized to a resource file which is broadcast to workers at run-time. This broadcasting may be a suboptimal solution if we want to embed the results of an earlier query, since these results can be processed more efficiently as a distributed collection. We believe that generating byte code at compile-time does not prevent us from implementing cost-based optimizations. Our plan, which we leave for a future work, is to generate conditional byte code at compile-time that considers many dynamic choices that a cost-based optimizer would consider at run-time. For join ordering, for example, the optimizer could select a few viable choices at compile-time based on greedy heuristics and generate conditional code that looks at statistical information available at run-time.

The contributions of this paper can be summarized as follows:

- We introduce a novel query language for large-scale, distributed data analysis, called DIQL, that is deeply embedded in Scala (Section III). Unlike other DISC query languages, the query checking and code generation are done at compile-time. With DIQL, programmers can express complex data analysis tasks, such as PageRank, k-means clustering, and matrix factorization, using SQL-like syntax exclusively.
- We present an algebra for DISC, called the monoid algebra, that captures most features supported by current DISC frameworks (Section IV), and rules for translating DIQL queries to the monoid algebra (Section V).
- We present algebraic transformations for deriving joins from nested queries that unnest nested queries of any form and any number of nesting levels, for pushing down predicates before joins, and for pruning unneeded data across operations, that generalize existing techniques (Section VI).
- We report on a prototype implementation of DIQL on three Big Data platforms, Spark, Flink, and Scalding (Section VII), and we show that DIQL has competitive performance relative to Spark DataFrames and Spark SQL (Section VIII).

II. Related Work

One of the earliest DISC frameworks was the Map-Reduce model, which was introduced by Google in 2004 [14]. The most popular Map-Reduce implementation is Apache Hadoop [5], an open-source project developed by Apache, which is used today by many companies to perform data analysis. Recent DISC systems go beyond Map-Reduce by maintaining dataset partitions in the memory of the worker nodes. Examples of such systems include Apache Spark [7] and Apache Flink [4]. There are also a number of higher-level languages that make Map-Reduce programming easier, such as HiveQL [32], PigLatin [26], SCOPE [12], DryadLINQ [33], and MRQL [17]. Apache Hive [32] 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. Apache Pig [19] provides a user-friendly scripting language, called PigLatin [26], on top of Map-Reduce, which allows explicit filtering, map, join, and group-by operations. Programs in PigLatin are written as a sequence of steps (a dataflow), where each step carries out a single data transformation. This sequence of steps is not necessarily executed in that order; instead, when a store operation is encountered, Pig optimizes the dataflow into a number of Map-Reduce barriers, which are executed as Map-Reduce jobs. Even though the PigLatin data model is nested, the language is less declarative than DIQL and does not support query nesting, but can simulate it using outer joins and coGroup. In addition to the DSLs for data-intensive programming, there are some Scala-based APIs that simplify Map-Reduce programming, such as, Scalding [27], which is part of Twitter's Cascading [11], Scrunch [28], which is a Scala wrapper for the Apache Crunch, and Scoobi. These

APIs support higher-order operations, such as map and filter, that are very similar to those for Spark and Flink. Slick [29] integrates databases directly into Scala, allowing stored and remote data to be queried and processed in the same way as in-memory data, using ordinary Scala classes and collections. Summingbird [10] is an API-based distributed system that supports run-time optimization and can run on both Map-Reduce and Storm. The main shortcoming of all these API-based approaches is their inability to analyze the functional arguments of their high-level operations at run-time to do complex optimizations.

Vertex-centric graph-parallel programming is a new popular framework for large-scale graph processing. It was introduced by Google's Pregel [25] but is now available by many open-source projects, such as Apache Giraph and Spark's GraphX. Most of these frameworks are based on the Bulk Synchronous Parallelism (BSP) programming model. VERTEXICA [24] and Grail [15] provide the same vertex-centric interface as Pregel but, instead of a distributed file system, they use a relational database to store the graph and the exchanged messages across the BSP supersteps. Unlike Grail, which can run on a single server only, VERTEXICA can run on multiple parallel machines connected to the same database server. Such configuration may not scale out very well because the centralized database may become the bottleneck of all the data traffic across the machines. Although DIQL is a general-purpose DISC query system, graph queries in DIQL are expressed using SQL-like syntax since graphs are captured as regular distributed collections. These queries are translated to distributed self-joins over the graph data. In [16], we proved that the monoid algebra can simulate any BSP computation (including vertex-centric parallel programs) efficiently, requiring the same amount of data shuffling as a typical BSP implementation.

DryadLINQ [33] is a programming model for large scale data-parallel computing that translates programs expressed in the LINQ programming model to Dryad, which is a distributed execution engine for data-parallel applications. Like DIQL, LINQ is a statically strongly typed language and supports a declarative SQL-like syntax. Unlike DIQL, though, the LINQ query syntax is very limited and has limited support for query nesting. DryadLINQ allows programmers to provide manual hints to guide optimization. Currently, it performs static optimizations based on greedy heuristics only, but there are plans to implement cost-based optimizations in the future.

Our work on embedded DSLs has been inspired by Emma ([2], [3]). Unlike our work, Emma does not provide an SQL-like query syntax; instead, it uses Scala's for-comprehensions 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. Scala's for-comprehensions are translated to monad comprehensions, which are desugared to monad operations, which, in turn, are expressed in terms of fold. Other 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 DIQL, 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 [10], which can run on both Map-Reduce and Storm. Compared to Spark and Flink, the Summingbird API is intentionally more restrictive to facilitate optimization at run-time. Summingbird though 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.

The syntax and semantics of DIQL have been influenced by previous work on list comprehensions with order-by and group-by [35]. Monoid homomorphisms and monoid comprehensions were first introduced as a formalism for OO queries in [18] and later as a query algebra for MRQL [17]. Our monoid algebra extends these algebras to include group-by, coGroup, and order-by operations. Many other algebras and calculi are based on monads and monad comprehensions. Monad comprehensions have a number of limitations, such as inability to capture grouping and sorting on bags. Monads also require various extensions to capture aggregations and heterogeneous operations, such as, converting a list to a bag. Monoids on the other hand can naturally capture aggregations, ordering, and group-by without any extension. Many other algebras and calculi in related work are based on monad comprehensions [34]. In an earlier work [16], we have shown that monoid comprehensions are a better formal basis for practical DISC query languages than monad comprehensions.

III. A DATA-INTENSIVE QUERY LANGUAGE FOR DISTRIBUTED DATA ANALYSIS

The syntax of DIQL is based on the syntax of MRQL [17], which is a query language for large-scale, distributed data analysis. The design of MRQL, in turn, has been influenced by XQuery and ODMG OQL, although it uses SQL-like syntax. Unlike MRQL, DIQL allows general Scala patterns and expressions in a query, it is more tightly integrated with the host language, and it is optimized and compiled to byte code at compile-time, instead of run-time.

Many emerging Scala-based APIs for distributed processing, such as the Scala-based Hadoop Map-Reduce frameworks Scalding and Scrunch, and the Map-Reduce alternatives Spark

pattern:	p	::=	*any Scala pattern, including a refutable pattern*	
qualifier:	q	::=	$p <- e$	*generator over a dataset*
		|	$p <-- e$	*generator over a small dataset*
		|	$p = e$	*binding*
qualifiers:	$\overline{q}$	::=	$q_1, \ldots, q_n$	*sequence of qualifiers*
aggregator:	$\oplus$	::=	$+ \mid * \mid \&\& \mid \mid\mid \mid \text{count} \mid \text{avg} \mid \text{min} \mid \text{max} \mid \ldots$	
expression:	e	::=	*any Scala functional expression*	
		|	**select** [**distinct**] e' **from** $\overline{q}$ [**where** e_p] [**group by** p [: e]] [cg] [**having** e_h]] [**order by** e_s]	*select-query*
		|	**repeat** $p = e$ **step** e' [**where** e_p] [**limit** n]	*repetition*
		|	**some** $\overline{q}$: e_p	*existential quantification*
		|	**all** $\overline{q}$: e_p	*universal quantification*
		|	**let** $p = e_1$ **in** e_2	*let-binding*
		|	e_1 opr e_2	opr *is member, union, intersect, or minus*
		|	$\oplus / e$	*aggregation*
co-group:	cg	::=	**from** $\overline{q}$ [**where** e_p] **group by** p [: e]	*the right branch of a coGroup*

Fig. 1. The DIQL Syntax

and Flink, are based on distributed collections that resemble regular Scala data collections as they support similar methods. Many Big Data analysis applications need to work on nested collections, because, unlike relational databases, they need to analyze data in their native format, as they become available, without having to normalize these data into flat relations first. While outer collections need to be distributed in order to be processed in parallel, the inner sub-collections must be stored in memory and processed as regular Scala collections. Processing both kinds of collections, distributed and in-memory, using the same syntax or API simplifies program development considerably. The DIQL syntax treats distributed and in-memory collections in the same way, although DIQL optimizes and compiles the operations on these collections differently. The DIQL data model supports four collection types: a bag, which is an unordered collection (a multiset) of values stored in memory, a DataBag, which is a bag distributed across the worker nodes of a computer cluster, a list, which is an ordered collection in memory, and a DataList, which is an ordered DataBag.

Consider the following Scala class that represents a graph node:

```
case class Node ( id: Long, adjacent: List[Long] )
```

where adjacent contains the ids of the node's neighbors. Then, a graph in Spark can be represented as a distributed collection of type RDD[Node]. Note that the inner collection of type List[Long] is an in-memory collection since it conforms to the class Traversable. The following DIQL query constructs the graph RDD from a text file stored in HDFS and then transforms this graph so that each node is linked with the neighbors of its neighbors:

```
q("""
    let graph = select Node( id = n, adjacent = ns )
        from line <- sc.textFile("graph.txt"),
             n::ns = line.split(",").toList
                         .map(_.toLong)
    in select Node( x, ++/ys )
       from Node(x,xs) <- graph,
            a <- xs,
            Node(y,ys) <- graph
       where y == a group by x
""")
```

The DIQL syntax can be embedded in a Scala program using the Scala macro q("""... """), which is optimized and compiled to byte code at compile-time, which in turn is embedded in the byte code generated by the rest of the Scala program. That is, all type errors in the DIQL queries are captured at compile-time. Furthermore, a query can see any active Scala declaration, including the results of other queries if they have been assigned to Scala variables. The DIQL syntax is Scala syntax extended with the DIQL select-query syntax (among other things), as is described in Figure 1. It uses a purely functional subset of Scala, which intentionally excludes blocks, declarations, iterations (such as while loops), and assignments, because imperative features are hard to optimize. Instead, DIQL provides special syntax for let-binding and iteration. DIQL queries can use any Scala pattern to pattern-match and deconstruct data. In the previous DIQL query, the let-binding binds the new variable graph to an RDD of type RDD[Node], which contains the graph nodes read from HDFS, mixing DIQL syntax with Spark API methods. The DIQL type-checker will infer that the type of sc.textFile("graph.txt") is RDD[String], and, hence, the type of the range variable line is String. Based on this information, the DIQL type-checker will infer that the type of n:ns in the pattern of the let-binding is List[Long], which is a Traversable collection. The second select-query (in the let-binding body) expresses a self-join over graph combined with a group-by. The pattern Node(x,xs)

matches one graph element at a time (a Node) and binds x to the node id and xs to the node adjacent list. Hence, the domain of the second qualifier a <- xs is an in-memory collection of type List[Long], making 'a' a variable of type Long. The graph is traversed a second time and its elements are matched with Node(y,ys). Thus, this select-query traverses both distributed and in-memory collections.

In general, as we can see in Figure 1, a select-query may have multiple qualifiers in the 'from' clause of the query. A qualifier $p <- e$, where e is a DIQL expression that returns a data collection and p is a Scala pattern (which can be refutable), iterates over the collection and, each time, the current element of the collection is pattern-matched with p, which, if the match succeeds, binds the pattern variables in p to the matched components of the element. The qualifier $p <-- e$ does the same iteration as $p <- e$ but it also gives a hint to the DIQL optimizer that the collection e is small (can fit in the memory of a worker node) so that the optimizer may consider using a broadcast join. The binding $p = e$ pattern-matches the pattern p with the value returned by e and, if the match succeeds, binds the pattern variables in p to the matched components of the e value. The variables introduced by a qualifier pattern can only be accessed by the remaining qualifiers and the rest of the query.

Unlike other query languages, patterns are essential to the DIQL group-by semantics; they fully determine which parts of the data are lifted to collections after the group-by operation. The group-by clause with syntax **group by** p:e groups the query result by the key calculated by the expression e. If e is missing, it is taken to be equal to p. For each group-by result, the pattern p is pattern-matched with the key, while every other pattern variable in the query (one that does not occur in p) is lifted to an in-memory collection that contains all the values of this pattern variable associated with this group-by result. For our example query, the query result is grouped by x, which is both the pattern p and the group-by expression e. After group-by, all pattern variables in the query except x, namely xs, y, and ys, are lifted to collections. In particular, ys of type List[Long] is lifted to a collection of type List[List[Long]], which is a list that contains all ys associated with a particular value of the group-by key x. Thus, the aggregation ++/ys will merge all values in ys using list append, ++, yielding a List[Long]. In general, the DIQL syntax $\oplus/e$ may use any Scala operation $\oplus$ of type $(T,T) \to T$ to reduce a collection of T (distributed or in-memory) to a value T. In addition, the same syntax can be used for the avg and count operations over a collection. The existential quantification **some** $\overline{q}$: e_p is syntactic sugar for $||/(\textbf{select}\ e_p\ \textbf{from}\ \overline{q})$, that is, it tests the predicate e_p for all elements in the collection resulted from the qualifiers $\overline{q}$ and checks if at least one result is true. Universal quantification does a similar operation using the aggregator &&.

The Scala code for the previous group-by query generated by DIQL is equivalent to a call to the Spark cogroup method that joins the graph with itself, followed by a reduceByKey:

```
graph.flatMap{ case n@Node(x,xs) => xs.map{ a => (a,n) } }
  .cogroup(graph.map{ case m@Node(y,ys) => (y,m) })
  .flatMap{ case (_,(ns,ms))
      => ns.flatMap{ case Node(x,xs)
            => xs.flatMap{ a => ms.map{ case Node(y,ys)
                                  => (x,ys) } } } }
  .reduceByKey(_++_).map{ case (x,s) => Node(x,s) }
```

The co-group clause cg represents the right branch of a coGroup operation (the left branch is the first group-by clause). As explained in the matrix addition example in the Introduction, each of the two group-by clauses lifts the pattern variables in their associated from-clause qualifiers and joins the results on their group-by keys. Hence, the rest of the query can access the group-by keys from the group-by clauses and the lifted pattern variables from both from-clauses.

The order-by clause of a select-query has syntax **order by** e_s, which sorts the query result by the sorting key e_s. One may use the pre-defined method desc to coerce a value to an instance of the class Inv, which inverts the total order of an Ordered class from $\leq$ to $\geq$. For example,

```
select p
from p@(x,y) <- S
order by ( x desc, y )
```

sorts the query result by major order x (descending) and minor order y (ascending).

Finally, to capture data analysis algorithms that require iteration, such as data clustering and PageRank, DIQL provides a general syntax for repetition:

$$\textbf{repeat}\ p\ =\ e\ \textbf{step}\ e'\ \textbf{where}\ e_p\ \textbf{limit}\ n$$

which does the assignment $p = e'$ repeatedly, starting with $p = e$, until either the condition e_p becomes false or the number of iterations has reached the limit n. In Scala, this is equivalent to the following code, where x is the result of the repetition:

```
var x = e
for (i <- 0 to n)
    x match { case p if e_p => x = e'
              case _ => break }
```

As we can see, the variables in p can be accessed in e', allowing us to calculate new variable bindings from the previous ones. The fixpoint type can be anything, including a tuple with distributed and in-memory collections, counters, etc. For example, the following query implements the k-means clustering algorithm by repeatedly deriving k new centroids from the previous ones:

```
repeat centroids = Array( ... )
step select Point( avg/x, avg/y )
     from p@Point(x,y) <- points
     group by k: ( select c from c <- centroids
                   order by distance(c,p) ).head
limit 10
```

where points is a distributed dataset of points on the X-Y plane of type RDD[Point], where Point is the Scala class:

```
case class Point ( X: Double, Y: Double ),
```

centroids is the current set of centroids (k cluster centers), and distance is a Scala method that calculates the Euclidean

distance between two points. The initial value of centroids (the ... value) is an array of k initial centroids. The inner select-query in the group-by clause assigns the closest centroid to a point p. 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. That is, the group-by query generates k groups, one for each centroid. For a group associated with a centroid c, the variable p is lifted to a Iterable[Point] that contains the points closest to c, while x and y are lifted to Iterable[Double] collections that contain the X- and Y-coordinates of these points. DIQL implements this query in Spark as a flatMap over points followed by a reduceByKey. The Spark reduceByKey operation does not materialize the Iterable[Double] collections in memory; instead, it calculates the avg aggregations in a stream-like fashion. DIQL caches the result of the repeat step, which is an RDD, into an array, because it has decided that centroids should be stored in an array (like its initial value). Furthermore, the flatMap functional argument accesses the centroids array as a broadcast variable, which is broadcast to all worker nodes before the flatMap.

IV. The Monoid Algebra

The main goal of our work is to translate DIQL 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 simplified if we first translate the queries to an algebraic form that is equivalent to the query and then translate the algebraic form to code consisting of calls to API operations supported by the underlying DISC platform. Our algebra consists of a very small set of operations that capture all the DIQL features and can be translated to efficient byte code that uses the Scala-based APIs of DISC platforms. We intentionally use only one higher-order homomorphic operation in our algebra, namely flatMap, to simplify normalization and optimization of algebraic terms. The flatMap operation captures data parallelism, where each processing node evaluates the same code (the flatMap 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 flatMap, our algebra detaches data distribution (specified by groupBy and coGroup) from parallel data processing (specified by flatMap). This separation simplifies program optimization considerably. For example, as we will see in Section VI-B, query unnesting can be done using just one rule, because there is only one place where a nested query can occur: at the flatMap functional argument. This section extends our previous work on the monoid algebra [16] by improving the semantics of some operations (such as, orderBy and repeat).

Collection monoids and homomorphisms. 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:

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

Monoids may satisfy 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 $x \in S$. 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. Given two monoids $\otimes$ and $\oplus$, a *monoid homomorphism* from $\otimes$ to $\oplus$ is a function H that respects the monoid structure:

$$\begin{aligned} H(X \otimes Y) &= H(X) \oplus H(Y) \\ H(\mathbf{1}_\otimes) &= \mathbf{1}_\oplus \end{aligned}$$

A data collection, such as a list, a set, or a bag, of type $T(\alpha)$, for some type α, can be captured as a *collection monoid* $\circledast$, which is a monoid equipped with a unit injection function $\mathbb{U}_\circledast$ of type $\alpha \to T(\alpha)$. The following table shows some well-known collection types captured as collection monoids:

	$\circledast$	$T(\alpha)$	$\mathbf{1}_\circledast$	$\mathbb{U}_\circledast(x)$	properties
list	$+\!\!+$	$[\alpha]$	$[\,]$	$[x]$	
bag	$\uplus$	$\{\!\{\alpha\}\!\}$	$\{\!\{\,\}\!\}$	$\{\!\{x\}\!\}$	commutative
set	$\cup$	$\{\alpha\}$	$\{\,\}$	$\{x\}$	commut. & idempotent

For example, $\{\!\{1\}\!\} \uplus \{\!\{2\}\!\} \uplus \{\!\{1\}\!\}$ constructs the bag $\{\!\{1, 2, 1\}\!\}$. A *collection homomorphism* is a monoid homomorphism $\mathcal{H}(\oplus, f)$ from a collection monoid $\otimes$ to a monoid $\oplus$ defined as follows, for $H = \mathcal{H}(\oplus, f)$:

$$\begin{aligned} H(X \otimes Y) &= H(X) \oplus H(Y) \\ H(\mathbb{U}_\otimes(x)) &= f(x) \\ H(\mathbf{1}_\otimes) &= \mathbf{1}_\oplus \end{aligned}$$

For example, $\mathcal{H}(+, \lambda x.\, 1)\, X$ over the bag X returns the number of elements in X. Not all collection homomorphisms are well-behaved; some may actually lead to contradictions. In general, a collection homomorphism from a collection monoid $\otimes$ to a monoid $\oplus$ is well-behaved if $\oplus$ satisfies all the laws that $\otimes$ does (the laws in our case are commutativity and idempotence). For example, converting a list to a bag is well-behaved, while converting a bag to a list and set cardinality are not. For example, set cardinality would have led to the contradiction: $2 = \mathcal{H}(+, \lambda x.\, 1)\, (\{a\} \cup \{a\}) = \mathcal{H}(+, \lambda x.\, 1)\, \{a\} = 1$.

All unary operations in the monoid algebra are collection homomorphisms. To capture binary equi-joins, we define binary functions that are homomorphic on both inputs. A *binary homomorphism* H from the collection monoids $\oplus$ and $\otimes$ to the monoid $\odot$ satisfies, for all X, X', Y, and Y':

$$H(X \oplus X', Y \otimes Y') = H(X, Y) \odot H(X', Y')$$

One such function is $H(X, Y) = \mathcal{H}(\odot, f_x)(X) \odot \mathcal{H}(\odot, f_y)(Y)$, for some functions f_x and f_y that satisfy:

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

The Monoid Algebra. As explained in Section III, the DIQL data model supports four collection types: a bag, which is an unordered collection (a multiset) of values stored in memory, a DataBag, which is a bag distributed across the worker nodes of a computer cluster, a list, which is an ordered collection in memory, and a DataList, which is an ordered DataBag. Both bags and lists are implemented as Scala Traversable collections, while both DataBags and DataLists are implemented as RDDs in Spark, DataSets in Flink, and TypedPipes in Scalding. The DIQL type inference engine can distinguish bags from lists and DataBags from DataLists by just looking at the algebraic operations: the orderBy operation always returns a list or a DataList, while the order is destroyed by all DIQL algebraic operations except flatMap. To simplify the presentation of our algebra, we present only one variation of each operation, namely one that works on bags. The flatMap, for example, has many variations: from a bag to a bag using a functional that returns a bag, from a list to list using a functional that returns a list, from a DataBag to a DataBag using a functional that returns a bag, etc, but some combinations are not permitted as they are not well-behaved, such as a flatMap from a bag to a list.

The flatMap operation. The first operation, flatMap, generalizes the select, project, join, cross, and unnest operations of the nested relational algebra. Given two arbitrary types α and β, the operation $\mathrm{flatMap}(f, X)$ over a bag X of type $\{\!\{\alpha\}\!\}$ applies the function f of type $\alpha \to \{\!\{\beta\}\!\}$ to each element of X, yielding one bag for each element, and then merges these bags to form a single bag of type $\{\!\{\beta\}\!\}$. It is a monoid homomorphism equivalent to the following equations:

$$\begin{array}{rcl} \mathrm{flatMap}(f, X \uplus Y) & = & \mathrm{flatMap}(f, X) \uplus \mathrm{flatMap}(f, Y) \\ \mathrm{flatMap}(f, \{\!\{a\}\!\}) & = & f(a) \\ \mathrm{flatMap}(f, \{\!\{\,\}\!\}) & = & \{\!\{\,\}\!\} \end{array}$$

Many common distributed queries can be written using flatMaps, including the equi-join $X \bowtie_p Y$:

$$\mathrm{flatMap}(\lambda x.\, \mathrm{flatMap}(\lambda y.\ \text{if } p(x,y) \text{ then } \{\!\{(x,y)\}\!\} \text{ else } \{\!\{\,\}\!\}, Y),\ X)$$

The flatMap operation is the only higher-order homomorphic operation in the monoid algebra. Its functional argument can be any Scala function, including a partial function with a refutable pattern; in that case, the second flatMap equation becomes a pattern matching that returns an empty bag if the refutable pattern fails to match.

The groupBy operation. Given a type κ that supports value equality ($=$), a type α, and a bag X of type $\{\!\{(\kappa, \alpha)\}\!\}$, the operation $\mathrm{groupBy}(X)$ groups the elements of X by their first component (the group-by key) and returns a bag of type $\{\!\{(\kappa, \{\!\{\alpha\}\!\})\}\!\}$ (a key-value map, also known as an indexed set). It is a monoid homomorphism equivalent to the following equations:

$$\begin{array}{rcl} \mathrm{groupBy}(X \uplus Y) & = & \mathrm{groupBy}(X) \Updownarrow \mathrm{groupBy}(Y) \\ \mathrm{groupBy}(\{\!\{(k,v)\}\!\}) & = & \{\!\{(k, \{\!\{v\}\!\})\}\!\} \\ \mathrm{groupBy}(\{\!\{\,\}\!\}) & = & \{\!\{\,\}\!\} \end{array}$$

The monoid $\Updownarrow$, called *indexed set union*, is a full outer join that merges groups associated with the same key using bag union. For example, $\mathrm{groupBy}(\{\!\{(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (1, \text{“}c\text{”})\}\!\})$ returns $\{\!\{(1, \{\!\{\text{“}a\text{”}, \text{“}c\text{”}\}\!\}), (2, \{\!\{\text{“}b\text{”}\}\!\})\}\!\}$.

The orderBy operation. Given a type κ that supports a total order ($\leq$), a type α, and a bag X of type $\{\!\{(\kappa, \alpha)\}\!\}$, the operation $\mathrm{orderBy}(X)$ returns a list of type $[(\kappa, \alpha)]$ sorted by the order-by key κ. It is a monoid homomorphism equivalent to the following equations:

$$\begin{array}{rcl} \mathrm{orderBy}(X \uplus Y) & = & \mathrm{orderBy}(X) \Uparrow \mathrm{orderBy}(Y) \\ \mathrm{orderBy}(\{\!\{(k,v)\}\!\}) & = & [(k,v)] \\ \mathrm{orderBy}(\{\!\{\,\}\!\}) & = & [\,] \end{array}$$

where the monoid $\Uparrow$ merges two sorted sequences of type $[(\kappa, \alpha)]$ to create a new sorted sequence.

The reduce operation. Aggregations are captured by the monoid homomorphism $\mathrm{reduce}(\oplus, X)$, which aggregates a bag X of type $\{\!\{\alpha\}\!\}$ using the monoid $\oplus$, returning a value of type α. For example, $\mathrm{reduce}(+, \{\!\{1,2,3\}\!\}) = 6$.

The coGroup operation. Although general joins and cross products can be expressed using nested flatMaps, we provide a special homomorphic operation that captures lossless equi-joins and outer joins. The operation $\mathrm{coGroup}(X, Y)$ between a bag X of type $\{\!\{(\kappa, \alpha)\}\!\}$ and a bag Y of type $\{\!\{(\kappa, \beta)\}\!\}$ over their first component of type κ (the join key) returns a bag of type $\{\!\{(\kappa, (\{\!\{\alpha\}\!\}, \{\!\{\beta\}\!\}))\}\!\}$. It is homomorphic on both inputs as it satisfies the law:

$$\begin{array}{l} \mathrm{coGroup}(X_1 \uplus X_2, Y_1 \uplus Y_2) \\ \quad = \ \mathrm{coGroup}(X_1, Y_1) \updownarrow \mathrm{coGroup}(X_2, Y_2) \end{array}$$

where the monoid $\updownarrow$ merges pairs of bags associated with the same key. For example, the coGroup operation over $\{\!\{(1,10),(2,20),(1,30)\}\!\}$ and $\{\!\{(1,40),(2,50),(3,60)\}\!\}$ is $\{\!\{(1, (\{\!\{10,30\}\!\}, \{\!\{40\}\!\})), (2, (\{\!\{20\}\!\}, \{\!\{50\}\!\})), (3, (\{\!\{\,\}\!\}, \{\!\{60\}\!\}))\}\!\}$.

The cross product operation, $\mathrm{cross}(X, Y)$ is the Cartesian product between X of type $\{\!\{\alpha\}\!\}$ and Y of type $\{\!\{\beta\}\!\}$ that returns $\{\!\{(\alpha, \beta)\}\!\}$. Unlike coGroup, it is not a monoid homomorphism:

$$\begin{array}{l} \mathrm{cross}(X_1 \uplus X_2, Y_1 \uplus Y_2) \\ \quad = \ \mathrm{cross}(X_1, Y_1) \uplus \mathrm{cross}(X_1, Y_2) \\ \qquad \uplus \mathrm{cross}(X_2, Y_1) \uplus \mathrm{cross}(X_2, Y_2) \end{array}$$

The repeat operation. Given a value X of type α, a function f of type $\alpha \to \alpha$, a predicate p of type $\alpha \to \mathrm{boolean}$, and a counter n, the operation $\mathrm{repeat}(f, p, n, X)$ returns a value of type α. It is defined as follows:

$$\begin{array}{rcl} \mathrm{repeat}(f, p, n, X) & \triangleq & \text{if } n \leq 0 \ \vee \ \neg p(X) \\ & & \text{then } X \\ & & \text{else } \mathrm{repeat}(f, p, n-1, f(X)) \end{array}$$

The repetition stops when the number of remaining repetitions is zero or the result value fails to satisfy the condition p.

V. Translation of DIQL to the Monoid Algebra

Figure 2 presents the rules for translating DIQL queries to the monoid algebra. The form $\mathcal{Q}[\![\, e \,]\!]$ translates the DIQL syntax of e (defined in Figure 1) to the monoid algebra. The

$$
\begin{array}{rcll}
\mathcal{Q}[\![\,\textbf{select distinct}\ e'\ \textbf{from}\ \ldots\,]\!] & = & \mathrm{flatMap}(\lambda(k,s).\,\{\!\{k\}\!\},\ \mathrm{groupBy}(\mathcal{Q}[\![\,\textbf{select}\ (e',())\ \textbf{from}\ \ldots\,]\!])) & (1)\\
\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ \ldots\ \textbf{order by}\ e\,]\!] & = & \mathrm{flatMap}(\lambda(k,v).\,\{\!\{v\}\!\},\ \mathrm{orderBy}(\mathcal{Q}[\![\,\textbf{select}\ (e,e')\ \textbf{from}\ \ldots\,]\!])) & (2)\\
\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ \overline{q}\ \textbf{group by}\ p:e\ \textbf{having}\ e_h\,]\!] & = & \mathrm{flatMap}(\lambda(p,s).\,\mathrm{lift}(\mathcal{V}^p_{\overline{q}},s,\mathrm{if}\ (\mathcal{Q}[\![\,e_h\,]\!])\ \mathrm{then}\ \{\!\{\mathcal{Q}[\![\,e'\,]\!]\}\!\}\ \mathrm{else}\ \{\!\{\,\}\!\}), & \\
 & & \quad \mathrm{groupBy}(\mathcal{Q}[\![\,\textbf{select}\ (e,\mathcal{V}^p_{\overline{q}})\ \textbf{from}\ \overline{q}\,]\!])) & (3)\\
\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ \overline{q}\ \textbf{group by}\ p:e\,]\!] & = & \mathrm{flatMap}(\lambda(p,s).\,\mathrm{lift}(\mathcal{V}^p_{\overline{q}},s,\{\!\{\mathcal{Q}[\![\,e'\,]\!]\}\!\}), & \\
 & & \quad \mathrm{groupBy}(\mathcal{Q}[\![\,\textbf{select}\ (e,\mathcal{V}^p_{\overline{q}})\ \textbf{from}\ \overline{q}\,]\!])) & (4)\\
\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ p<\!-\,e,\,\overline{q}\,]\!] & = & \mathrm{flatMap}(\lambda p.\,\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ \overline{q}\,]\!],\ \mathcal{Q}[\![\,e\,]\!]) & (5)\\
\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ p<\!-\!-\,e,\,\overline{q}\,]\!] & = & \mathrm{flatMap}(\lambda p.\,\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ \overline{q}\,]\!],\ \mathrm{small}(\mathcal{Q}[\![\,e\,]\!])) & (6)\\
\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ p=e,\,\overline{q}\,]\!] & = & \mathrm{let}\,p=\mathcal{Q}[\![\,e\,]\!]\,\mathrm{in}\,\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ \overline{q}\,]\!] & (7)\\
\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from where}\ e_p\,]\!] & = & \mathrm{if}\ (\mathcal{Q}[\![\,e_p\,]\!])\ \mathrm{then}\ \{\!\{\mathcal{Q}[\![\,e'\,]\!]\}\!\}\ \mathrm{else}\ \{\!\{\,\}\!\} & (8)\\
\mathcal{Q}[\![\,\textbf{select}\ e'\ \textbf{from}\ \,]\!] & = & \{\!\{\mathcal{Q}[\![\,e'\,]\!]\}\!\} & (9)\\
\mathcal{Q}[\![\,\textbf{repeat}\ p\ =\ e\ \textbf{step}\ e'\ \textbf{where}\ e_p\ \textbf{limit}\ n\,]\!] & = & \mathrm{repeat}(\lambda p.\,\mathcal{Q}[\![\,e'\,]\!],\ \lambda p.\,\mathcal{Q}[\![\,e_p\,]\!],\ n,\ \mathcal{Q}[\![\,e\,]\!]) & (10)\\
\mathcal{Q}[\![\,\textbf{some}\ \overline{q}:\ e_p\,]\!] & = & \mathrm{reduce}(||,\ \mathcal{Q}[\![\,\textbf{select}\ e_p\ \textbf{from}\ \overline{q}\,]\!]) & (11)\\
\mathcal{Q}[\![\,\textbf{all}\ \overline{q}:\ e_p\,]\!] & = & \mathrm{reduce}(\&\&,\ \mathcal{Q}[\![\,\textbf{select}\ e_p\ \textbf{from}\ \overline{q}\,]\!]) & (12)\\
\mathcal{Q}[\![\,\oplus/e\,]\!] & = & \mathrm{reduce}(\oplus,\mathcal{Q}[\![\,e\,]\!]) & (13)\\
\mathcal{Q}[\![\,e_1\ \textbf{intersect}\ e_2\,]\!] & = & \mathcal{Q}[\![\,\textbf{select}\ x\ \textbf{from}\ x<\!-X\ \textbf{where some}\ y<\!-Y:x==y\,]\!] & (14)
\end{array}
$$

Fig. 2. Query Translation Rules

notation $\lambda p.e$, where p is a Scala pattern, is an anonymous function f defined as $f(p)=e$. Although the rules in Figure 2 translate join queries to nested flatMaps, in Section VI-B, we present a general method for deriving joins from nested flatMaps.

The order that select-queries are translated in Figure 2 is as follows: the distinct clause is translated first, then the order-by clause, then the group-by/having clauses, then the 'where' clause, then the 'from' clause (the query qualifiers), and finally the select header (the query result). Rule (1) translates a distinct query to a group-by that removes duplicates by grouping the query result values by the entire value. Rule (2) translates an order-by query to an orderBy operation.

Rules (3) and (4) translate group-by queries. Recall that, after group-by, every pattern variable in a query except the group-by pattern variables, are lifted to a collection of values to contain all bindings of this variable associated with a group. To specify these bindings, we use the notation $\mathcal{V}^p_{\overline{q}}$ to denote the flat tuple that contains all pattern variables in the sequence of qualifiers $\overline{q}$ that do not appear in the group-by pattern p. It is defined as $\mathcal{V}^p_{\overline{q}}=(v_1,\ldots,v_n)$ (the order of $v_1,\ldots,v_n$ is unimportant), where $v_i\in(\mathcal{V}[\![\,\overline{q}\,]\!]-\mathcal{P}[\![\,p\,]\!])$ and $\mathcal{V}$ is defined as follows:

$$
\begin{array}{rcl}
\mathcal{V}[\![\,p<\!-\,e,\,\overline{q}\,]\!] & = & \mathcal{P}[\![\,p\,]\!]\cup\mathcal{V}[\![\,\overline{q}\,]\!]\\
\mathcal{V}[\![\,p<\!-\!-\,e,\,\overline{q}\,]\!] & = & \mathcal{P}[\![\,p\,]\!]\cup\mathcal{V}[\![\,\overline{q}\,]\!]\\
\mathcal{V}[\![\,p=e,\,\overline{q}\,]\!] & = & \mathcal{P}[\![\,p\,]\!]\cup\mathcal{V}[\![\,\overline{q}\,]\!]
\end{array}
$$

where $\mathcal{P}[\![\,p\,]\!]$ is the set of pattern variables in the pattern p. As a simplification, we lift only the variables that are accessed by the rest of the query, since all others are ignored. The query **select** $(e,\mathcal{V}^p_{\overline{q}})$ **from** $\overline{q}$ is the query before the group-by operation that returns a collection of pairs that consist of the group-by key and the non-group-by variables $\mathcal{V}^p_{\overline{q}}$. Hence, after groupBy, the non-group-by variables are grouped into a collection to be used by the rest of the query. The operation $\mathrm{lift}((v_1,\ldots,v_n),\,s,e)$, where s is the collection that contains the values of the non-group-by variables in the current group, lifts each variable v_i to a collection by rebinding it to a collection derived from s as $\mathrm{flatMap}(\lambda(v_1,\ldots,v_n).\,\{\!\{v_i\}\!\},\,s)$. Rule (5) translates a generator $p<\!-e$ to a flatMap. Rule (6) is similar, but embeds the annotation 'small' (which is the identity operation), to be used by the optimizer as a hint about the collection size. Rule (7) gives the translation of a let-binding for an irrefutable pattern p. For a refutable p, the translation is:

$$
\begin{array}{c}
\mathcal{Q}[\![\,e\,]\!]\ \mathrm{match}\,\{\,\mathrm{case}\,p\ \Rightarrow\ \mathcal{Q}[\![\,\textbf{select}\ e\ \textbf{from}\ \overline{q}\,]\!];\\
\mathrm{case}\,\_\ \Rightarrow\ \{\!\{\,\}\!\}\,\}
\end{array}
$$

Rules (8) and (9) are the last rules to be applied when all qualifiers in the query have been translated, and hence, the 'from' clause is empty. Rule (8) translates the 'where' clause of a query with an empty 'from' clause. Rule (10) translates the repeat syntax to the repeat operation. Finally, Rule (13) translates accumulations to reduce operations. Finally, Rule (14) translates bag intersection to a select query. There are similar rules for bag membership and difference.

For example, in the matrix multiplication query presented in the Introduction, the only variable that needs to be lifted after group-by is z, since it is the only one used by the rest of the query. That is, $\mathcal{V}^p_{\overline{q}}=(z)$. Hence, the matrix multiplication

translated as follows:

$$
\begin{aligned}
&\mathcal{Q}[\![\ \textbf{select}\ (+/z,i,j)\\
&\quad \textbf{from}\ (x,i,k)\!<\!-X,\ (y,k\_,j)\!<\!-Y,\ z=x*y\\
&\quad \textbf{where}\ k==k\_\ \textbf{group by}\ (i,j)\ ]\!]\\
&=\text{flatMap}(\lambda((i,j),s).\ \text{let}\ z=\text{flatMap}(\lambda z.\ \{\!\{z\}\!\},\ s)\\
&\qquad\qquad\qquad \text{in}\ \{\!\{\mathcal{Q}[\![\ (+/z,i,j)\ ]\!]\}\!\}\\
&\qquad\quad \text{groupBy}(\mathcal{Q}[\![\ \textbf{select}\ ((i,j),z)\\
&\qquad\qquad\qquad\quad \textbf{from}\ (x,i,k)\!<\!-X,\ (y,k\_,j)\!<\!-Y,\\
&\qquad\qquad\qquad\qquad z=x*y\ \textbf{where}\ k==k\_\ ]\!])\\
&=\text{flatMap}(\lambda((i,j),s).\ \{\!\{(\text{reduce}(+,s),i,j)\}\!\},\\
&\qquad \text{groupBy(join))}
\end{aligned}
$$

where join is:

$$
\begin{aligned}
&\mathcal{Q}[\![\ \textbf{select}\ ((i,j),z)\\
&\quad \textbf{from}\ (x,i,k)\!<\!-X,\ (y,k\_,j)\!<\!-Y,\ z=x*y\\
&\quad \textbf{where}\ k==k\_\ ]\!]\\
&=\text{flatMap}(\lambda(x,i,k).\\
&\qquad \text{flatMap}(\lambda(y,k\_,j).\\
&\qquad\qquad \text{let}\ z=x*y\\
&\qquad\qquad \text{in if}\ (k==k\_)\ \text{then}\ \{\!\{((i,j),z)\}\!\}\\
&\qquad\qquad\qquad\qquad \text{else}\ \{\!\{\ \}\!\},\\
&\qquad\qquad Y),\ X)\\
&=\text{flatMap}(\lambda(x,i,k).\\
&\qquad \text{flatMap}(\lambda(y,k\_,j).\\
&\qquad\qquad \text{if}\ (k==k\_)\ \text{then}\ \{\!\{((i,j),x*y)\}\!\}\\
&\qquad\qquad\qquad\qquad \text{else}\ \{\!\{\ \}\!\},\\
&\qquad\qquad Y),\ X)
\end{aligned}
$$

We will see in Section VI-B that join-like queries like this, expressed with nested flatMaps, are optimized to coGroups.

Finally, Figure 2 does not show the rules for translating a co-group clause cg defined in Figure 1. These double group-by clauses are translating to coGroups using the following rule:

$$
\begin{aligned}
&\mathcal{Q}[\![\ \textbf{select}\ e'\ \textbf{from}\ \overline{q_1}\ \textbf{group by}\ p_1:e_1\\
&\qquad\qquad \textbf{from}\ \overline{q_2}\ \textbf{group by}\ p_2:e_2\ ]\!]\\
&=\text{flatMap}(\lambda(p,(s_1,s_2)).\,\text{lift}(\mathcal{V}^{p_1}_{\overline{q_1}},s_1,\text{lift}(\mathcal{V}^{p_2}_{\overline{q_2}},s_2,\{\!\{\mathcal{Q}[\![\ e'\ ]\!]\}\!\})),\\
&\qquad \text{coGroup}(\mathcal{Q}[\![\ \textbf{select}\ (e_1,\mathcal{V}^{p_1}_{\overline{q_1}})\ \textbf{from}\ \overline{q_1}\ ]\!],\\
&\qquad\qquad\quad \mathcal{Q}[\![\ \textbf{select}\ (e_2,\mathcal{V}^{p_2}_{\overline{q_2}})\ \textbf{from}\ \overline{q_2}\ ]\!]))
\end{aligned}
$$

VI. Optimization

Given a DIQL query, our goal is to translate it into an algebraic form that can be evaluated efficiently in a DISC platform. Current database technology has already addressed this problem in the context of relational databases. The DIQL data model and algebra though are richer than those of SQL, requiring special optimization techniques that are better suited for a DISC framework. Currently, our optimizations are not based on any cost model; instead, they are heuristics that take into account the hints provided by the user, such as using a $<\!-\!-$ qualifier to traverse a small dataset. That is, our framework tries to identify all possible joins and convert them to broadcast joins if they are over a small dataset, but currently, it does not optimize the join order. In this section, we present the normalization rules that put DIQL terms to a canonical form and a general method for query unnesting. Two more optimization techniques are given in the Appendix: pushing down predicates (Appendix A) and column pruning (Appendix A).

A. Query Simplification

Cascaded flatMaps can be fused into a single nested flatMap using the following law that is well-known in functional PLs:

$$
\begin{aligned}
&\text{flatMap}(f,\text{flatMap}(g,S))\\
&\qquad\qquad \to \text{flatMap}(\lambda x.\,\text{flatMap}(f,g(x)),S) \qquad (15)
\end{aligned}
$$

If S is a distributed dataset, this normalization rule reduces the number of required distributed operations from two to one and eliminates redundant operations if we apply further optimizations to the inner flatMap. It generalizes well-known algebraic laws in relational algebra, such as fusing two cascaded selections into one. If we apply this transformation repeatedly, any algebraic term can be normalized into a tree of groupBy, coGroup, and cross operations with a single flatMap between each pair of these operations. There are many other standard normalization rules, such as projecting over a tuple, $(e_1,\ldots,e_n).\_i = e_i$, and rules that are derived directly from the operator's definition, such as $\text{flatMap}(f,\ \{\!\{a\}\!\}) = f(a)$.

B. Deriving Joins and Unnesting Queries

The translation rules from the DIQL syntax to the monoid algebra, presented in Section V, translate join-like queries to nested flatMaps, instead of coGroups. In this section, we present a general method for identifying any possible equi-join from nested flatMaps, including joins across deeply nested queries. (An equi-join is a join between two datasets X and Y whose join predicate takes the form $k_1(x) = k_2(y)$, for $x \in X$ and $y \in Y$, for some key functions k_1 and k_2.) It is precisely 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 flatMaps to coGroups is crucial for good performance in distributed processing. Without joins, the only way to evaluate a nested flatMap, such as $\text{flatMap}(\lambda x.\,\text{flatMap}(\lambda y.\,h(x,y),\ Y),\ X)$, in a distributed environment, where both collections X and Y are distributed, would be 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 flatMap to a coGroup, we create opportunities for more evaluation strategies, which may include the broadcast solution. For example, one good evaluation strategy for large X and Y that are 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 (called a distributed partitioned join). While related approaches for query unnesting ([18], [21]) require many rewrite rules to handle various cases of query nesting, our method requires only one rule and is more general as it handles nested queries of any form and any number of nesting levels.

Consider the following DIQL query Q over X and Y:

```
Q = select b
    from (a,b) <− X
    where b > +/(select d from (c,d) <− Y where a==c)
```

One efficient method for evaluating this nested query is to first group Y by its first component and aggregate:

```
Z = select (c,+/d) from (c,d) <− Y group by c
```

and then to join X with Z:

```
Q' = select b
     from (a,b) <− X, (c,sum) <− Z
     where a==c && b>sum
```

This query though is not equivalent to the original query Q; if an element of X has no matches in Y, then this element will appear in the result of Q (since the sum over an empty bag is 0), but it will not appear in the result of Q' (since it does not satisfy the join condition). To correct this error, Q' should use a left-outer join between X and Z. In other words, using the monoid algebra, we want to translate query Q to:

$$\begin{array}{l}\text{flatMap}(\lambda(k,(xs,ys)).\,\text{flatMap}(\lambda b.\\ \qquad\text{if } b > \text{reduce}(+,ys) \text{ then } \{\!\{b\}\!\} \text{ else } \{\!\{\,\}\!\},\ xs)\\ \quad\text{coGroup}(\text{flatMap}(\lambda(a,b).\,\{\!\{(a,b)\}\!\},\ X),\\ \qquad\text{flatMap}(\lambda(c,d).\,\{\!\{(c,d)\}\!\},\ Y)))\end{array}$$

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 now generalize the previous example to convert nested flatMaps to joins. We consider patterns of algebraic terms of the form $\text{flatMap}(\lambda p_1.\, g(\text{flatMap}(\lambda p_2.\, e,\, e_2)), e_1)$, for some patterns p_1 and p_2, some terms e_1, e_2, and e, and some term function g. (A term function is an algebraic term that contains its arguments as subterms.) For the cases we consider, the term e_2 should not depend on the pattern variables in p_1, otherwise it would not be a join. This algebraic term matches most pairs of nested flatMaps on bags that are equivalent to joins, including those derived from nested queries, such as the previous DIQL query, and those derived from flat join-like queries. Thus, the method presented here detects and converts any possible join (implicit or explicit) to a coGroup.

We consider the following term function as a join candidate:

$$F(X,Y) = \text{flatMap}(\lambda p_1.\, g(\text{flatMap}(\lambda p_2.\, e,\, Y)), X) \quad (16)$$

We require that $g(\{\!\{\,\}\!\}) = \{\!\{\,\}\!\}$ so that $F(X,Y) = \{\!\{\,\}\!\}$ if either X or Y is empty. To transform $F(X,Y)$ to a join between X and Y, we need to derive two terms k_1 and k_2 from e (these are the join keys), such that $k_1 \neq k_2$ implies $e = \{\!\{\,\}\!\}$, and k_1 depends on the p_1 variables while k_2 depends on the p_2 variables, exclusively. Then, if there are such terms k_1 and k_2, we transform $F(X,Y)$ to the following join $F'(X,Y)$:

$$\begin{array}{l}\text{flatMap}(\lambda(k,(xs,ys)).\,F(xs,ys),\\ \quad\text{coGroup}(\text{flatMap}(\lambda x@p_1.\,\{\!\{(k_1,x)\}\!\},\ X),\\ \qquad\text{flatMap}(\lambda y@p_2.\,\{\!\{(k_2,y)\}\!\},\ Y)))\end{array} \quad (17)$$

(Recall that the Scala pattern binder $x@p$ matches a value with the pattern p and binds x to that value.) The proof that $F'(X,Y) = F(X,Y)$ is given in Theorem A.1 in the Appendix. This is the only transformation rule needed to derive any possible join from a query and unnest nested queries because there is only one higher-order operator in the monoid algebra (flatMap) that can contain a nested query.

For example, the nested query Q presented at the beginning of this section is translated to the following algebraic term $Q(X,Y)$:

$$\begin{array}{l}\text{flatMap}(\lambda(a,b).\,\text{if } (b > \text{reduce}(+,\text{flatMap}(\lambda(c,d).\\ \qquad\qquad\text{if } (a == c) \text{ then } \{\!\{d\}\!\} \text{ else } \{\!\{\,\}\!\},\ Y))\\ \qquad\text{then } \{\!\{b\}\!\} \text{ else } \{\!\{\,\}\!\},\ X)\end{array}$$

This term matches the term function $F(X,Y)$ in Equation (16) using $g(z) = \text{if } (b > \text{reduce}(+,z)) \text{ then } \{\!\{b\}\!\} \text{ else } \{\!\{\,\}\!\}$ and $e = \text{if } (a == c) \text{ then } \{\!\{d\}\!\} \text{ else } \{\!\{\,\}\!\}$. We can see that $k_1 = a$ and $k_2 = c$ because $a \neq c$ implies $e = \{\!\{\,\}\!\}$. Thus, from Equation (17), $Q(X,Y)$ is transformed to:

$$\begin{array}{l}\text{flatMap}(\lambda(k,(xs,ys)).\,Q(xs,ys),\\ \quad\text{coGroup}(\text{flatMap}(\lambda x@(a,b).\,\{\!\{(a,x)\}\!\},\ X),\\ \qquad\text{flatMap}(\lambda y@(c,d).\,\{\!\{(c,y)\}\!\},\ Y)))\end{array}$$

The biggest challenge in applying the transformation in Equation (17) is to derive the join keys, k_1 and k_2, from the term e. If flatMaps are normalized using Equation (15) first, then the only place these key functions can appear is in a flatMap functional argument. Then, the body of the flatMap functional argument can be another flatMap, in which case we consider the latter flatMap, or a term **if** p **then** e' **else** $\{\!\{\,\}\!\}$, in which case we derive the keys from p, such that $k_1 \neq k_2 \Rightarrow \neg p$. If p is already in the form $k_1 == k_2$, then, in addition to deriving the keys k_1 and k_2, we can simplify the term $F(xs,ys)$ in Equation (17) by replacing the term **if** p **then** e' **else** $\{\!\{\,\}\!\}$ with e'.

Finally, the term $F'(X,Y)$ in Equation (17) assumes that we can derive a pair of keys, k_1 and k_2, such that k_1 depends only on p_1, and k_2 depends only on p_2. But, often, such keys do not exists because they depend on variable bindings introduced inside the term function g and the term e. In such cases, calculating the keys before coGroup using $\text{flatMap}(\lambda x@p_1.\,\{\!\{(k_1,x)\}\!\},\ X)$ and $\text{flatMap}(\lambda y@p_2.\,\{\!\{(k_2,y)\}\!\},\ Y)$, would be incorrect because the terms k_1 and k_2 depend on additional variables. To compensate, we collect all operations in g and e that introduce new bindings, namely flatMaps, into two corrective term functions, m_x and m_y, to bind the variables used by the keys k_1 and k_2, respectively. More specifically, the coGroup in Equation (17) will become:

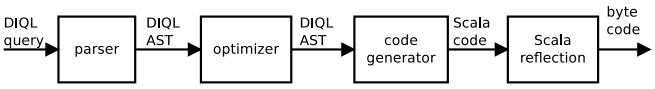
$$\begin{array}{l}\text{coGroup}(\text{flatMap}(\lambda x@p_1.\,m_x(\{\!\{(k_1,x)\}\!\}),\ X),\\ \qquad\text{flatMap}(\lambda y@p_2.\,m_y(\{\!\{(k_2,y)\}\!\}),\ Y))\end{array}$$

Starting with an initial set of variables $V_x = \mathcal{P}[\![\, p_1 \,]\!]$ and $m_x = \lambda x.\, x$ for X, and $V_y = \mathcal{P}[\![\, p_2 \,]\!]$ and $m_y = \lambda y.\, y$ for Y, we consider every $\text{flatMap}(\lambda p.\, b,\ e)$ in the terms g and e that leads to the keys k_1 and k_2. If the term e depends

on the variables in V_x only, then $V_x \leftarrow V_x \cup \mathcal{P}[\![p]\!]$ and $m_x \leftarrow \lambda x.\, \mathrm{flatMap}(\lambda p.\, m_x(x),\ e)$. If the term e depends on the variables in V_y only, then $V_y \leftarrow V_y \cup \mathcal{P}[\![p]\!]$ and $m_y \leftarrow \lambda y.\, \mathrm{flatMap}(\lambda p.\, m_y(y),\ e)$. Otherwise, if the term e depends on the variables in both V_x and V_y, then it is impossible to derive a coGroup because we cannot calculate each one of the keys separately.

VII. Implementation

DIQL generates Scala code, which is translated to JVM byte code at compile-time using Scala's compile-time reflection facilities. This code generation is vastly simplified with the use of quasiquotes, which allow one to construct internal Scala abstract syntax trees (ASTs) by just embedding Scala syntax in quotes. After a DIQL query is parsed, it is translated to an AST. DIQL ASTs are a subset of Scala's ASTs for expressions and patterns, since they need to capture Scala's purely functional syntax only, but are extended with special nodes to capture our algebraic operators. The following figure shows the query translation process at compile time:



As explained in Section III, the DIQL data model extends Scala's types with four collection types: bag, list, DataBag, and DataList. The DIQL code generator needs to know which collection types are used by the algebraic operations to decide which API to use for each operation. In addition, many optimizations apply only when the operations involved are over distributed datasets, which requires knowledge about the types. To determine the kind of collections used by the algebraic operations, the DIQL type inference system uses the Scala type checker. All Scala Traversable objects (that is, instances of classes that conform to the type Traversable, such as a List) are in-memory collections (bags or lists). Instances of distributed collection classes supported by the underlying distributed framework are distributed collections (DataBags or DataLists). For Spark, for example, instances of classes that conform to the class RDD (such as, an RDD or any subclass of RDD) are taken to be distributed collections. Furthermore, the DIQL type inference engine distinguishes bags from lists and DataBags from DataLists by just looking at the algebraic operations: the orderBy operation always returns a list or a DataList, while the order is destroyed by all monoid algebraic operations except flatMap. A bag (resp. a DataBag) is more restricted than a list (resp. a DataList) as it does not permit certain operations that depend on order, such as indexing.

In Section IV, to simplify the presentation, we present only one variation of each algebraic operation, namely one that works on bags. The flatMap, for example, has many variations: from a bag to a bag using a functional that returns a bag, from a list to list using a functional that returns a list, from a DataBag to a DataBag using a functional that returns a bag, etc, but some combinations are not permitted as they are not well-behaved, such as a flatMap from a bag to a list.

The DIQL type inference system extends the Scala type inference system using a very simple trick: DIQL generates Scala code and the code is type-checked using Scala's compile-time reflection facilities. More specifically, the DIQL type-checker uses a typing environment that binds Scala patterns to Scala types. When type-checking a DIQL algebraic term e under a typing environment that consists of the bindings $p_1 : t_1, \ldots, p_n : t_n$, which bind the patterns p_i to their types t_i, it calls the DIQL code generator to generate a Scala AST c from the term e and then it calls the Scala type-checker to get the type of the Scala code $(p_1 : t_1) \Rightarrow \cdots (p_n : t_n) \Rightarrow c$, which will derive a functional type $t_1 \rightarrow \cdots t_n \rightarrow t$, for some type t, provided that the code c is type-correct. This type t is then the type of e. The typing environment is extended during the type-checking of a DIQL query and of the algebraic operators derived from the query. In Spark, for instance, the type t of the term e_2 in $\mathrm{flatMap}(\lambda p.\, e_1,\, e_2)$ must conform to the class RDD$[t']$ or to the class Traversable$[t']$, for some type t'. Then the functional body e_1 is type-checked using the current typing environment extended with the binding $p : t'$. Note that DIQL queries are type-checked before optimization and code generation by translating them using the rules in Fig. 2 and type-checking the resulting terms.

The DIQL optimizer finds all possible joins using the method in Section VI-B, then finds all possible cross products, then applies the rest of the optimizations described in Appendix A, then factors out all common terms (so that their value is cached at run-time), and finally generates broadcast operations for the in-memory operations that are accessed in the functional parameter of a flatMap over a DataBag. CoGroups and cross products between a DataBag and a 'small' DataBag (as defined by the $<$— qualifier) or an in-memory collection, are translated to broadcast joins and cross products.

Not all nested flatMaps on DataBags can be translated to coGroups and cross products. For example, DIQL evaluates the following query on the DataBags S and R:

```
select (x,+/(select w from (z,w) <- R where z<y))
from (x,y) <- S
```

by broadcasting R to all worker nodes.

DIQL abstracts a DISC platform using an abstract class that is parameterized by a type constructor DataBag (these are called higher kinded types in Scala, and are type constructors that take type constructors as type parameters):

```
abstract class DistrCodeGen[DataBag[_]]
```

which defines an abstract method for each DIQL operator:

```
def flatMap[A,B]( f: (A) => Traversable[B], S: DataBag[A] )
                (implicit tag: ClassTag[B]): DataBag[B]
```

where the implicit tag is required by all DISC Scala APIs to allow run-time reflection. For example, the Spark code generator binds DataBag to RDD:

```
class SparkCodeGenerator extends DistrCodeGen[RDD]
```

and implements the abstract methods based on RDDs. That way, the DIQL code generator is modularized and is very easy

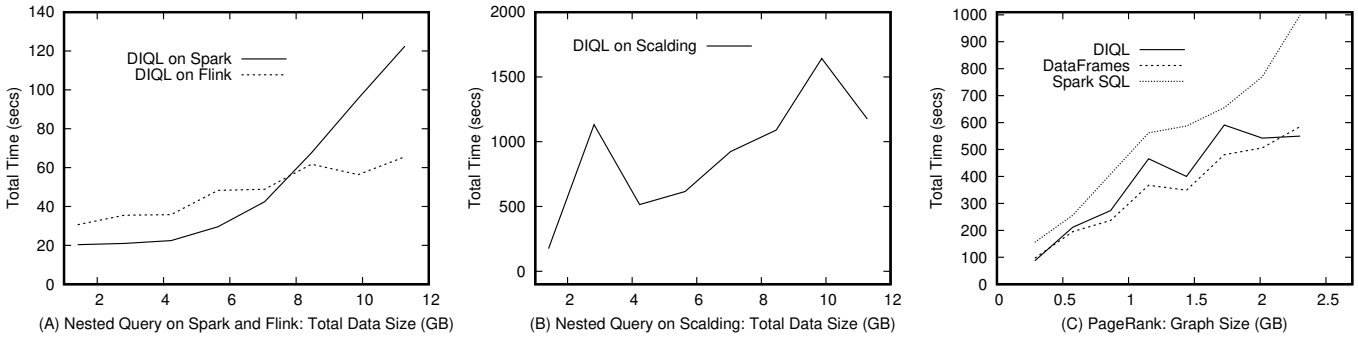


Fig. 3. Nested Query and PageRank Evaluation

to extend to multiple DISC platforms. Out of the 4560 lines of the DIQL implementation, the code generator for Spark is 468 lines, for Flink is 570 lines, and for Scalding is 386 lines.

VIII. Performance Evaluation

The DIQL source code is available at https://github.com/fegaras/DIQL. This web site includes a directory, benchmarks, that contains all the source files and scripts for running the experiments reported in this section. The first set of experiments is to evaluate a nested query on the three different platforms supported by DIQL. The purpose of these experiments is to show that DIQL can efficiently evaluate a nested query that cannot be expressed as such in Hive and Spark SQL. The second set of experiments is to compare the DIQL system with the Spark DataFrames and Spark SQL frameworks [30] by evaluating PageRank on various graph sizes. The purpose of these experiments is to show that DIQL has competitive performance relative to Spark DataFrames and Spark SQL, although it does not use cost-based optimizations.

The nested query used in our first set of evaluations is very similar to the one used as a running example in Section VI-B:

```
select c.name from c <− customers
where c.account < +/(select o.price from o <− orders
                     where o.cid == c.cid)
```

It finds all customers whose account is less than the total price of their orders. Our system translates this query to a simple coGroup, which is implemented as a distributed partitioned join. Hive and Spark SQL do not support this kind of query nesting (they only support IN and EXIST subqueries in predicates), thus requiring to express this query as a left outer join. The PageRank query, as well as the DataFrames and Spark SQL programs, used in our second set of evaluations are given in Appendix A.

The platform used for our evaluations is a small cluster built on the XSEDE Comet cloud computing infrastructure at the SDSC (San Diego Supercomputer Center). Each Comet node has 24 Xeon E5 cores at 2.5GHz, 128GB RAM, and 320GB of SSD for local scratch memory. For our experiments, we used Apache Hadoop 2.6.0, Apache Spark 2.1.0 running in standalone mode, Apache Flink 1.2.0 running on Yarn, and Scalding 0.17.0 in Map-Reduce mode on Yarn. The HDFS file system was formatted with the block size set to 128MB and the replication factor set to 3. Each experiment was evaluated 4 times under the same data and configuration parameters. Each data point in the plots in Fig. 3 represents the mean value of 4 experiments.

The Customer and Order datasets used in our first experiments contained random data. We used 8 pairs of datasets Customer-i and Order-i, for $i = 1 \ldots 8$, where Customer-i has $i * 4 * 10^6$ tuples and Order-i has $i * 4 * 10^7$ tuples so that each customer is associated with an average of 10 orders. The total size of Customer-i and is Order-i is $i * 1.41$ GB. That is, the largest input has a total size 11.28GB. For the nested query evaluation, we used 4 Comet nodes. The results of the nested query evaluation are shown in Fig. 3.A for Spark and Flink, and in Fig. 3.B for Scalding on Map-Reduce. These two figures have been separated because both Spark and Flink are many times faster than Hadoop Map-Reduce used in Scalding. From these figures, we can see that the previous nested query is evaluated efficiently as a regular join, since it is translated to a coGroup. Other DISC query systems, such as Hive and Spark SQL, support very simple forms of query nesting in predicates, which does not include the evaluated query, thus forcing the programmers to use explicit outer joins to simulate the other forms of nested queries and write explicit code to handle nulls.

The graphs used in our PageRank experiments were synthetic data generated by the RMAT Graph Generator [13] using the Kronecker 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 8 datasets of size $i * 288$MB, with $i * 2 * 10^6$ vertices and $i * 2 * 10^7$ edges, for $i \in [1, 8]$. That is, the largest dataset used was 2.25GB. The PageRank algorithm was evaluated using DIQL, Spark DataFrames, and Spark SQL. For the PageRank evaluation, we used 10 Comet nodes, which were organized into 42 Spark executors, where each executor had 5 cores and 24GB memory. That is, these 42 executors used a total of $42*5 = 210$ cores and 1TB memory. Each dataset used in our experiments was stored in HDFS in $42 * 2 = 84$ partitions (HDFS files). The spark.sql.shuffle.partitions in DataFrames was set to 84 to match the number of files. The results of the PageRank evaluation are shown in Figure 3. We can see that the DIQL

run time is between the run times of the DataFrames code (best) and the Spark SQL query (worst).

IX. Conclusion and Future Work

We have presented a powerful SQL-like query language for data analysis that is deeply embedded in Scala and a query optimization framework that generates efficient byte code at compile-time. In our quest to minimize the impedance mismatch between query and host programming languages, we have found compile-time reflection to be a very valuable tool. It simplifies query validation by giving access to the host type-checking engine, thus seamlessly integrating the host with the query binding environments. This integration is harder to achieve at run-time because some information on declarations and types is lost at run-time, and can only be recovered if we reconstruct parts of the symbol table at run-time. We have also found that quasiquotes and macros simplify code generation because they allow us to generate byte code using the host language syntax, instead of constructing host ASTs, which are often complex and may change in the future, or using a special code generation language, such as LLVM, which is harder to integrate with the host binding environment. Furthermore, the code generated from quasiquotes is highly optimized by the host compiler and may be of better quality than that of LLVM since it is specific to the host language. Unfortunately, currently, very few programming languages support compile-time reflection and macros, with the notable exceptions of Scala, Haskell, and F#. On the other hand, code generators, such as LLVM, can by used by most programming languages.

Currently, DIQL is a prototype system with plenty of room for improvements. As a future work, we are planning to implement cost-based optimizations by generating optimization choices statically to be evaluated dynamically based on statistics. In addition to the traditional database optimizations, we are planning to introduce platform-specific cost-based optimizations, such as adjusting the number of reducers in operations that cause data shuffling (such as join and group-by) based on the total number of available workers and the generated workflow. Finally, DIQL allows API code to be mixed with the query syntax but treats this API code as a black box. As a future work, we will translate API method calls to the monoid algebra so that they too can be optimized along with the rest of the query.

References

[1] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. An Embedded DSL for High Performance Big Data Processing. In *BigData'12*.

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

[3] A. Alexandrov, , *et al*. Emma in Action: Declarative Dataflows for Scalable Data Analysis. In *SIGMOD'16*.

[4] Apache Flink. http://flink.apache.org/. 2017.

[5] Apache Hadoop. http://hadoop.apache.org/. 2017.

[6] Apache Hive. http://hive.apache.org/. 2017.

[7] Apache Spark. http://spark.apache.org/. 2017.

[8] M. Armbrust, *et al*. Spark SQL: Relational Data Processing in Spark. In *SIGMOD'15*.

[9] D. Battre, , *et al*. 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.

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

[11] Cascading: Application Platform for Enterprise Big Data http://www.cascading.org/, 2017.

[12] R. Chaiken, *et al*. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *PVLDB*, 1(2): 1265–1276, 2008.

[13] 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.

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

[15] J. Fan, *et al*. The case against specialized graph analytics engines. In *CIDR*, 2015.

[16] L. Fegaras. An Algebra for Distributed Big Data Analytics. Journal of Functional Programming, special issue on Programming Languages for Big Data, Volume 27, 2017.

[17] 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.

[18] L. Fegaras and D. Maier. Optimizing Object Queries Using an Effective Calculus. In *ACM Transactions on Database Systems (TODS)*, 25(4):457–516, 2000.

[19] A. F. Gates, *et al*. Building a High-Level Dataflow System on top of Map-Reduce: the Pig Experience. In *PVLDB*, 2(2): 1414-1425, 2009.

[20] 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.

[21] J. Holsch, M. Grossniklaus, and M. H. Scholl. Optimization of Nested Queries using the NF^2 Algebra. In *SIGMOD'16*.

[22] M. Isard, , *et al*. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *EuroSys*, 2007.

[23] M. Isard and Y. Yu. Distributed Data-Parallel Computing Using a High-Level Programming Language. In *SIGMOD'09*.

[24] A. Jindal, *et al*. Vertexica: Your Relational Friend for Graph Analytics! In *PVLDB*, 7(13): 1669–1672, 2014.

[25] G. Malewicz, *et al*. Pregel: a System for Large-Scale Graph Processing. In *Principles of Distributed Computing (PODC)*, 2009.

[26] C. Olston, , *et al*. Pig Latin: a not-so-Foreign Language for Data Processing. In *SIGMOD'08*.

[27] Scalding: Building Map-Reduce Applications with Scala. http://www.cascading.org/projects/scalding/

[28] Scrunch: A Scala Wrapper for the Apache Crunch Java API. 2017. http://crunch.apache.org/scrunch.html.

[29] Slick: Functional Relational Mapping for Scala. http://slick.lightbend.com/.

[30] Spark SQL, DataFrames, and Datasets Guide. 2017. http://spark.apache.org/docs/latest/sql-programming-guide.html.

[31] A. K. Sujeeth, *et al*. Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages. In *ACM Transactions on Embedded Computing Systems*, 13(4s) article 134, 2014.

[32] A. Thusoo, *et al*. Hive: a Warehousing Solution over a Map-Reduce Framework. In *PVLDB*, 2(2): 1626–1629, 2009.

[33] Y. Yu, *et al*. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[34] P. Wadler. Comprehending Monads. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, pp 61–78, 1990.

[35] P. Wadler and S. Peyton Jones. Comprehensive Comprehensions (Comprehensions with 'Order by' and 'Group by'). In *Haskell Symposium*, 2007.

[36] M. Zaharia, *et al*. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

APPENDIX

APPENDIX A: CORRECTNESS PROOF OF THE JOIN DERIVATION ALGORITHM

Theorem A.1 (Nested flatMaps to coGroup): Let k_1 and k_2 be two subterms of the term e such that $k_1 \neq k_2$ implies $e = \{\!\{\,\}\!\}$, then the term function:

$$F(X,Y) = \text{flatMap}(\lambda p_1.\, g(\text{flatMap}(\lambda p_2.\, e,\, Y)), X)$$

is equal to:

$$\begin{aligned}&\text{flatMap}(\lambda(k,(xs,ys)).\, F(xs,ys),\\ &\quad \text{coGroup}(\text{flatMap}(\lambda x@p_1.\, \{\!\{(k_1,x)\}\!\},\, X),\\ &\quad\quad \text{flatMap}(\lambda y@p_2.\, \{\!\{(k_2,y)\}\!\},\, Y)))\end{aligned}$$

provided that $g(\{\!\{\,\}\!\}) = \{\!\{\,\}\!\}$, k_1 depends only on p_1, and k_2 depends only on p_2.

Proof: First, it is easy to prove that

$$\begin{aligned}X = \text{flatMap}(\lambda(k,(xs,ys)).\, \text{flatMap}(\lambda x.\, \{\!\{(k,x)\}\!\},\, xs),\\ \text{coGroup}(X,Y))\end{aligned} \tag{18}$$

$$\begin{aligned}Y = \text{flatMap}(\lambda(k,(xs,ys)).\, \text{flatMap}(\lambda y.\, \{\!\{(k,y)\}\!\},\, ys),\\ \text{coGroup}(X,Y))\end{aligned} \tag{19}$$

The coGroup inputs are $X' = \text{flatMap}(\lambda x@p_1.\, \{\!\{(k_1,x)\}\!\}, X)$ and $Y' = \text{flatMap}(\lambda y@p_2.\, \{\!\{(k_2,y)\}\!\}, Y)$. Then,

$$\begin{aligned}&F(X,Y) = \text{flatMap}(\lambda p_1.\, g(\text{flatMap}(\lambda p_2.\, e,\, Y)), X)\\ &= \text{flatMap}(\lambda(k,p_1).\, g(\text{flatMap}(\lambda(k',p_2).\, e,\, Y')), X')\end{aligned}$$

From Equations (18) and (15), the outer flatMap becomes:

$$\begin{aligned}&\text{flatMap}(\lambda(k,p_1).\, g(G),\, X')\\ &= \text{flatMap}(\lambda(k,p_1).\, g(G),\\ &\quad \text{flatMap}(\lambda(k,(xs,ys)).\, \text{flatMap}(\lambda x.\, \{\!\{(k,x)\}\!\},\, xs),\\ &\quad\quad \text{coGroup}(X',Y')))\\ &= \text{flatMap}(\lambda(k,(xs,ys)).\, \text{flatMap}(\lambda x@p_1.\, g(G),\, xs),\\ &\quad \text{coGroup}(X',Y'))\end{aligned}$$

where G, from Equations (19) and (15), is:

$$\begin{aligned}&G = \text{flatMap}(\lambda(k',p_2).\, e,\, Y')\\ &= \text{flatMap}(\lambda(k',p_2).\, e,\\ &\quad \text{flatMap}(\lambda(k,(xs,ys)).\, \text{flatMap}(\lambda y.\, \{\!\{(k,y)\}\!\},\, ys),\\ &\quad\quad \text{coGroup}(X',Y')))\\ &= \text{flatMap}(\lambda(k',(xs',ys')).\, \text{flatMap}(\lambda y@p_2.\, e,\, ys'),\\ &\quad \text{coGroup}(X',Y'))\end{aligned}$$

which is equal to $\text{flatMap}(\lambda y@p_2.\, e,\, ys)$ because, for $k \neq k'$, they are both equal to $\{\!\{\,\}\!\}$ since $e = \{\!\{\,\}\!\}$, and for $k = k'$, we have $xs = xs'$ and $ys = ys'$, since they come from the same input $\text{coGroup}(X',Y')$. ■

APPENDIX B: THE PAGERANK CODE

We used the following Scala case classes to represent the graph edges and the PageRank entries:

```
case class Edge ( src: Int , dest: Int )
case class Rank ( id: Int , degree: Int , rank: Double )
```

The DIQL query for PageRank used in our evaluations is as follows:

```
let edges = select Edge(s,d)
            from line <- sc.textFile( input_file ),
                 List(s,d) = line.split(",").toList.map(_.toInt)
in repeat nodes = select Rank( id = s,
                               degree = (count/d).toInt,
                               rank = 1-alpha )
                  from Edge(s,d) <- edges
                  group by s
   step select Rank( id, m.degree, nr )
        from (id,nr) <- ( select ( key,
                                   (1-alpha)+alpha*(+/rank)/(+/degree) )
                          from Rank(id,degree,rank) <- nodes,
                               e <- edges
                          where e.src == id
                          group by key: e.dest ),
             m <- nodes
        where id == m.id
   limit 10
```

where alpha = 0.85 is the damping factor. The equivalent DataFrames program used in our experiments is as follows:

```
def rank ( sums: Long, counts: Long ): Double
         = (1-alpha)+alpha*sums/counts
val edges = spark.sparkContext.textFile( input_file )
                 .map(_.split(",")).map(n => Edge(n(0).toInt,n(1).toInt))
                 .toDF()
var nodes = edges.groupBy("src").agg(count("dest").as("degree"))
                 .withColumn("rank",lit(1-alpha))
                 .withColumnRenamed("src","id")
for ( i <- 1 to 10 ) {
  val newranks = nodes.join(edges,nodes("id")===edges("src"))
                      .groupBy("dest").agg(udf(rank _)
                      .apply(sum("rank"),sum("degree")).as("newrank"))
  nodes = newranks.join(nodes,nodes("id")===newranks("dest"))
                  .drop("rank","dest")
                  .withColumnRenamed("newrank","rank")
}
```

The Spark SQL code that is equivalent to the above DataFrames program initializes the nodes to the result of the following query:

```
SELECT src as id, count(dest) as count, 0.15 as rank
FROM edges
GROUP BY src
```

and then, inside a loop, reassigns nodes to the result of the following query:

```
SELECT n.id, n.count, 0.15+0.85*m.rank as rank
FROM nodes n
     JOIN ( SELECT e.dest, sum(n.rank/n.count) as rank
            FROM nodes n JOIN edges e ON n.id=e.src
            GROUP BY e.dest ) m
     ON m.dest=n.id
```

Initially, after the first few iterations, both the DataFrames and the Spark SQL programs were filtering most of the PageRank data from the nodes and were returning very few results. The problem turned out to be that it was very hard to force

DataFrames to cache the fixpoint of the iteration (nodes) in memory. Instead, DataFrames accumulated all operations from all iteration steps, and, since nodes was used twice at each iteration, there was an explosion of operations that were pipelined to be executed when the final result was written to HDFS. The resulting plan had bugs, which was understandable since it consisted of hundreds of operations. Applying cache() to nodes did not help because, unlike Spark, cache() is lazy in DataFrames. Also, applying an action, such as count(), to nodes did not help either, because DataFrames use stored statistics to calculate these aggregations. Our solution was to convert the nodes to an RDD at each iteration step, cache the RDD, and then convert this RDD back to a DataFrame.

APPENDIX C: OTHER OPTIMIZATIONS

Pushing Down Predicates

One important optimization technique used in relational databases is pushing selections before joins. We have generalized this technique to apply to complex data and queries. Given a predicate c and a set of variables V, $\text{split}[\![\, c \,]\!]_V$ returns a pair (c_1, c_2) such that $c_1 \wedge c_2 \Rightarrow c$ and c_1 may depend on the variables in V while c_2 must not.

The first optimization is pushing predicates before a groupBy:

$$\begin{aligned}&\text{flatMap}(\lambda(k,s).\, g(\text{flatMap}(\lambda p.\, \text{if } c \text{ } h(\text{then } e' \text{ else } \{\!\{\,\}\!\},\ s))),\\ &\qquad \text{groupBy}(e))\\ &= \text{flatMap}(\lambda(k,s).\, g(\text{flatMap}(\lambda p.\, h(\text{if } c_2 \text{ then } e' \text{ else } \{\!\{\,\}\!\},\ s))),\\ &\qquad \text{groupBy}(\text{flatMap}(\lambda(k, x@p).\, \text{if } c_1 \text{ then } \{\!\{x\}\!\} \text{ else } \{\!\{\,\}\!\},\\ &\qquad\qquad e)))\end{aligned}$$

where g and h are term functions with $g(\{\!\{\,\}\!\}) = h(\{\!\{\,\}\!\}) = \{\!\{\,\}\!\}$. Here, $(c_1, c_2) = \text{split}[\![\, c \,]\!]_V$ where $V = \mathcal{P}[\![\, p \,]\!] \cup \{k\}$, that is, c_1 may depend on k or the variables in p only. As in the join derivation algorithm, if g and h introduce new bindings through flatMaps, the flatMap before groupBy should be wrapped with a corrective term function to bind the variables used by these flatMaps.

Pushing a predicate into the left input of a join is done using the following transformation (a similar one exists for the right input):

$$\begin{aligned}&\text{flatMap}(\lambda(k,(xs,ys)).\\ &\qquad\qquad g(\text{flatMap}(\lambda p.\, h(\text{if } c \text{ then } e' \text{ else } \{\!\{\,\}\!\},\ xs)),\\ &\qquad \text{coGroup}(e_1, e_2))\\ &= \text{flatMap}(\lambda(k,(xs,ys)).\\ &\qquad\qquad g(\text{flatMap}(\lambda p.\, h(\text{if } c_2 \text{ then } e' \text{ else } \{\!\{\,\}\!\},\ xs)),\\ &\qquad \text{coGroup}(\text{flatMap}(\lambda(k, x@p).\, \text{if } c_1 \text{ then } \{\!\{x\}\!\} \text{ else } \{\!\{\,\}\!\},\\ &\qquad\qquad e_1),\\ &\qquad\quad e_2))\end{aligned}$$

where, here, c_1 may depend on k or the variables in p only.

Data Pruning

Column pruning is an important optimization for relational data-base queries because it reduces the size of intermediate results between operations. It is even more important for DISC frameworks since it reduces the amount of shuffled data, which is the prominent cost factor for distributed processing. We have developed a general data pruning method that removes unneeded data from complex data, which may be nested in arbitrary ways. Earlier work on complex data pruning [1] has used a holistic approach to analyze Spark programs written in Scala to identify unneeded data components and to transform these programs in such a way that they prune these data early. Unlike this earlier work, our work uses compositional rules to introduce a flatMap before each groupBy and coGroup that prunes only the parts of the data that are not used by the next operation. This newly introduced flatMap is then fused with the existing flatMap in the groupBy or coGroup input. By applying this transformation top-down, starting from the query result, we minimize the amount of shuffled data, one operation at a time.

We consider terms of the form $\text{flatMap}(f, \text{groupBy}(e))$. Terms with a coGroup can be handled in a similar way. We want to embed a flatMap before the groupBy that prunes all the data not used by the function f. Let $\{\!\{(k, tp)\}\!\}$ be the type of e. We want to find all the components of the type tp that are not accessed by the function f (the group-by key of type k cannot be pruned). We are using a heuristic method that works well for most cases. We construct a simple instance of the type tp, using $\mathcal{I}[\![\, tp \,]\!]$, defined as follows:

$$\begin{aligned}\mathcal{I}[\![\, \{\!\{t\}\!\} \,]\!] &= \{\!\{\mathcal{I}[\![\, t \,]\!]\}\!\}\\ \mathcal{I}[\![\, (t_1, \ldots, t_n) \,]\!] &= (\mathcal{I}[\![\, t_1 \,]\!], \ldots, \mathcal{I}[\![\, t_n \,]\!])\\ \mathcal{I}[\![\, t \,]\!] &= v_t \qquad \text{a basic type}\end{aligned}$$

where v_t is a new variable of type t. There are more cases that are not included here. If we optimize the term $f((k, \mathcal{I}[\![\, \{\!\{tp\}\!\} \,]\!]))$ using the normalization rules presented in Section VI-A, then, if the variable v_t associated with a type t does not occur in the resulting normalized term, then it is not accessed by f. In that case, we call this type t an unused type. For example, let

$$f = \lambda(k,s).\, \text{reduce}(+, \text{flatMap}(\lambda(x,y).\{\!\{y\}\!\},\ s))$$

and let the type tp be (int, int). Then, $\mathcal{I}[\![\, \{\!\{tp\}\!\} \,]\!] = \{\!\{(v_1, v_2)\}\!\}$, where v_1 corresponds to the first int and v_2 to the second. The normalized term $f((k, \{\!\{(v_1, v_2)\}\!\}))$ is $\text{reduce}(+, \{\!\{v_2\}\!\})$, which means that only the second int from the type (int, int) is used.

Based on the type annotations on the terms that represent basic types in tp (used or unused), we annotate all the subterms of the term tp: a type $\text{List}[t]$ is unused if t is unused, and a type $(t_1, \ldots, t_n)$ is unused if all $t_1, \ldots, t_n$ are unused. Given that every subterm in tp is marked as used or unused, we generate code (a function) that prunes all unused data from the input:

$$\begin{aligned}\mathcal{C}_1[\![\, t \,]\!] &= \lambda x.\, () \qquad \text{if } t \text{ is unused}\\ \mathcal{C}_1[\![\, \{\!\{t\}\!\} \,]\!] &= \lambda x.\, \text{flatMap}(\lambda z.\, \{\!\{\mathcal{C}_1[\![\, t \,]\!](z)\}\!\},\ x)\\ \mathcal{C}_1[\![\, (t_1, \ldots, t_n) \,]\!] &= \lambda(v_1, \ldots, v_n).\, (\mathcal{C}_1[\![\, t_{i_1} \,]\!](v_{i_1}), \ldots,\\ &\qquad\qquad \mathcal{C}_1[\![\, t_{i_m} \,]\!](v_{i_m}))\\ &\quad \text{for the used types } t_{i_1}, \ldots, t_{i_m}\\ \mathcal{C}_1[\![\, t \,]\!] &= \lambda x.\, x \qquad \text{otherwise}\end{aligned}$$

and a similar function $\mathcal{C}_2[\![\, t \,]\!]$ (not shown) that converts the pruned data to a t value by embedding the value $()$ in the place of the unused components, so that $\mathcal{C}_1[\![\, t \,]\!](\mathcal{C}_2[\![\, t \,]\!]\, x) = x$, for every x of type t. Then, $\mathcal{C}_1$ is used to deeply prune data before the groupBy, while $\mathcal{C}_2$ is used to insert $()$ values after groupBy:

$$\begin{aligned}
&\mathrm{flatMap}(f, \mathrm{groupBy}(e)) \\
&= \mathrm{flatMap}(f, \mathrm{flatMap}(\lambda(k,s).\, \{\!\{(k, \mathcal{C}_2[\![\, \{\!\{tp\}\!\} \,]\!]\, s)\}\!\}, \\
&\qquad\qquad \mathrm{groupBy}(\mathrm{flatMap}(\lambda(k,x).\, \{\!\{(k, \mathcal{C}_1[\![\, tp \,]\!]\, x)\}\!\}, \\
&\qquad\qquad\qquad e))))
\end{aligned}$$

After the top two flatMaps are fused using Equation 15, these $()$ values are eliminated since they are not used by f.

Based on the previous example, $\mathcal{C}_1((int, int)) = \lambda(v_1, v_2).\, v_2$ and $\mathcal{C}_2(\{\!\{(int, int)\}\!\}) = \lambda x.\, \mathrm{flatMap}(\lambda v_2.\, \{\!\{((), v_2)\}\!\},\, x)$. Thus, the data pruning is $\mathrm{flatMap}(\lambda(k,x).\, \{\!\{(k, \mathcal{C}_1[\![\, tp \,]\!]\, x)\}\!\},\, e)$ before groupBy, which is equal to $\mathrm{flatMap}(\lambda(k,(v_1,v_2)).\, \{\!\{(k, v_2\}\!\},\, e)$, for $x = (v_1, v_2)$. That is, it prunes the first column before groupBy.