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.**

***Keywords*-Big Data; Distributed Query Processing; Query Optimization; Embedded Query Languages;**

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 [10] 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 [18] and Scrunch, 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 [17], MRQL [12], 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 [19]. 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.

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.

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.

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 relational 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 [22] 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 report on a prototype implementation of DIQL on three Big Data platforms, Spark, Flink, and Scalding (Section IV) and explain how the DIQL type inference system is integrated with the Scala typing system using Scala's compile-time reflection facilities and macros.
- We give evidence that DIQL has competitive performance relative to Spark DataFrames and Spark SQL by evaluating a simple nested query and a PageRank query (Section V).

II. Related Work

One of the earliest DISC frameworks was the Map-Reduce model, which was introduced by Google in 2004 [10]. 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 [21], PigLatin [17], SCOPE, DryadLINQ [22], and MRQL [12]. Apache Hive [21] 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 [14] provides a user-friendly scripting language, called PigLatin [17], 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 [18], which is part of Twitter's Cascading [9], Scrunch, 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.

DryadLINQ [22] 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.

III. A Data-Intensive Query Language for Distributed Data Analysis

The syntax of DIQL is based on the syntax of MRQL [12], 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 and Flink, are based on distributed collections that resemble regular Scala data collections as they support similar methods. 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

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:	$\oplus$	::=	$+ \mid * \mid \&\& \mid \|\| \mid$ count $\mid$ avg $\mid$ min $\mid$ 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

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 $(T, T) \rightarrow T$ 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 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. 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
```

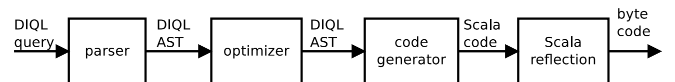
where points is a distributed dataset of points on the X-Y plane of type RDD[Point], where Point is the Scala class:

```
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. 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:



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

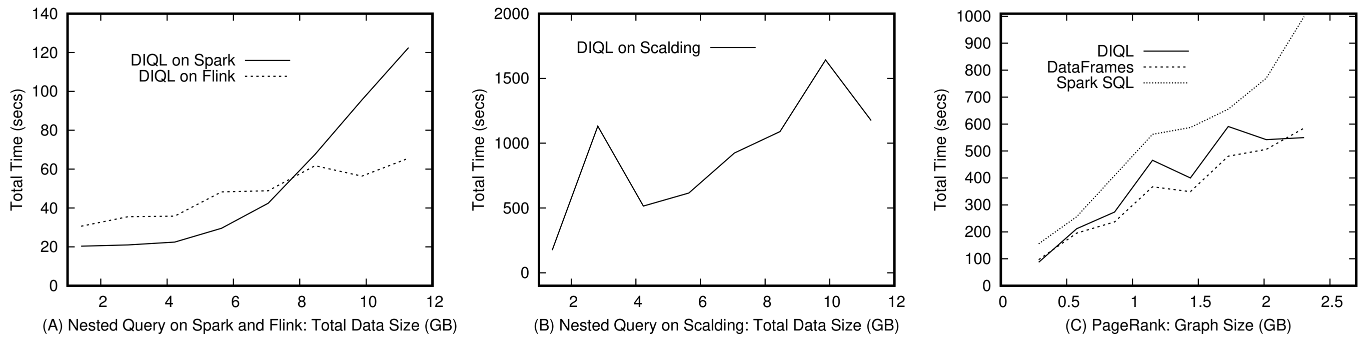


Fig. 2. Nested Query and PageRank Evaluation

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.

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.

The DIQL optimizer finds all possible joins and cross products, optimizes the algebraic terms, 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. A coGroup between a distributed DataBag and a 'small' DataBag (as defined by the $<\!\!\!\text{---}$ qualifier) or an in-memory bag is translated to a broadcast join, which broadcasts the small dataset to all workers.

V. 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 [19] 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.

Our first set of evaluations uses the following nested query:

```
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 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. 2 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. 2.A for Spark and Flink, and in Fig. 2.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.

The graphs used in our PageRank experiments were synthetic data generated by the RMAT Graph Generator 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∗288MB, 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 2. We can see that the DIQL run time is between the run times of the DataFrames code (best) and the Spark SQL query (worst).

VI. 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 existing relational 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] Cascading: Application Platform for Enterprise Big Data http://www.cascading.org/, 2017.

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

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

[12] 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.

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

[14] 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.

[15] 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.

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

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

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

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

[20] 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.

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

[22] 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.

[23] 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.