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

Leonidas Fegaras & Md Hasanuzzaman Noor

University of Texas at Arlington

July 2, 2018

Outline

- Limitations of current query languages for Big Data analysis
- Introducing DIQL: the Data Intensive Query Language
- The monoid algebra
- Converting complex nested queries to joins

Programming Environments for Big Data Analysis

- Designed for large-scale data processing on computer clusters
- Often work on raw data
- They hide the details of distributed computing and fault tolerance
- Many provide functional-style APIs
  - using powerful higher-order operations as building blocks
  - preventing interference among parallel tasks
- But some programmers are unfamiliar with functional programming
- They need to specify low-level details to optimize performance
- Too many competing platforms:

  Map-Reduce, Spark, Flink, Storm, ...
- Hard to tell which one will prevail in the near future

Solution:

Use a query language that is independent of the underlying platform!

Limitations of Current Query Languages for Big Data

They are subsets of SQL.
But raw data is often nested, not normalized.

Most query languages

- provide a limited syntax for operating on data collections
  simple joins and group-bys
- have limited support for nested collections and hierarchical data
- cannot express complex data analysis tasks that need iteration
- they treat in-memory and distributed collections differently

Wish List for Query Embedding

PL: host programming language, QL: embedded query language

- No *impedance mismatch*:
  a QL must be fully embedded into the host PL
- The QL and PL data models must be equivalent
  - . . . but a QL must also work on distributed collections that have special semantics
- Must use the same QL syntax to query both distributed and in-memory collections
  - although they may be processed differently
- No null values in the QL data model
  - we don't want to use 3-valued logic, like SQL
  - PLs do not provide a standardized way to treat nulls
    - nulls must be removed before UDF calls
  - $\Rightarrow$ no outer joins

DIQL (Data-Intensive Query Language)

An SQL-like query language for Big Data analysis that

- is deeply embedded in Scala
- is designed to support multiple Scala-based APIs for Big Data
  currently: Spark, Flink, and Twitter's Cascading/Scalding
- can uniformly work on both distributed and in-memory collections using the same syntax
- allows seamless mixing of native Scala code with SQL-like query syntax
  - can use any Scala pattern, access any Scala variable, and embed any pure Scala code
  - can use the core Scala libraries and user-defined classes
- has compositional semantics based on monoid homomorphisms

DIQL (Data-Intensive Query Language)

DIQL queries are optimized and translated to Java byte code at compile-time using Scala's compile-time reflection and macros

The optimizer can still do cost-based optimizations:

1. can pick few viable choices for query plans at compile-time, and
2. generate conditional code that chooses a plan at run-time

The DIQL Syntax

pattern: p ::= *any Scala pattern*

qualifier: q ::= $p \leftarrow e$ *generator over a dataset*
| $p \leftarrow\!\!- e$ *generator over a small dataset*
| $p = e$ *binding*

expression: e ::= *any pure Scala expression*
| **select** [**distinct**] e
from $q, \ldots, q$
[**where** e]
[**group by** p [: e] [**having** e]]
[**order by** e]
| $\oplus/e$ *aggregation*

The Group-By Semantics

- Aggregation is detached from group-by
- A group-by generates nested collections
- Pattern variables are essential to the group-by semantics
- A group-by lifts each pattern variable defined in the **from**-clause from some type t to a bag $\{t\}$

  this $\{t\}$ contains all the variable values associated with the same group-by key

Example: Matrix Multiplication

- A sparse matrix M is a bag of (v, i, j), for $v = M_{ij}$
- Matrix multiplication of X and $Y = \sum_k X_{ik} * Y_{kj}$:

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

- before the group-by: `z` $= X_{ik} * Y_{kj}$
- after group-by:
  - `z` is lifted to a bag of values $X_{ik} * Y_{kj}$;
    that is, for each group `(i, j)`, the bag `z` contains $X_{ik} * Y_{kj}$, for all k
  - the group-by variables `i` and `j` are not lifted to bags
- `+/z` sums up all `z` values, for each group `(i, j)`

How can we Express Outer Joins in DIQL?

Outer semijoins can be simply expressed as nested queries

In SQL:

```
select c.name
from Customers c left outer join Orders o on o.cid = c.cid
group by c.cid
having o.price is null or c.account >= sum(o.price)
```

in DIQL:

```
select c.name from c <- Customers
where c.account >= +/(select o.price from o <- Orders
                      where o.cid == c.cid)
```

The DIQL query processor converts most nested queries to coGroups

Mixing SQL-like Syntax with Scala Code

A Scala class that represents a graph node:

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

In Spark, a graph is a distributed collection of type RDD[Node].

Query: transform a graph so that each node is linked to the neighbors of its neighbors:

```
q("""
let graph = select Node( n, 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
""")
```

Monoids as a Formal Basis for Distributed Data-Centric Systems

- The results of data-parallel computations must be independent of
  - the way we divide the data into partitions and
  - the way we combine the partial results to obtain the final result

  $\Rightarrow$ data-parallel computations must be *associative*
- They can be expressed as *monoid homomorphisms*
  - A monoid has an associative merge function and an identity
  - A collection monoid has also a unit function

    $$(\uplus, \{\,\}, \lambda x.\{x\}) \qquad \text{for bags}$$
  - A monoid homomorphism maps a collection monoid to a monoid
  - Captures data parallelism

    $$H(P_1 \uplus P_2 \uplus \cdots \uplus P_n) = H(P_1) \oplus H(P_2) \oplus \cdots \oplus H(P_n)$$

    for some monoid $\oplus$

The Monoid Algebra

Generalizes the nested relational algebra

- **flatMap** applies a function to each bag element and merges the results:

$$\text{flatMap}(\lambda x.\, \{x + 1\},\ \{1, 2, 3\}) \ = \ \{2, 3, 4\}$$

  Monoid: $\uplus$

- **groupBy** returns an indexed set (a key-value map):

$$\text{groupBy}(\{(1, a), (2, b), (1, c), (1, d)\}) \ = \ \{(1, \{a, c, d\}), (2, \{b\})\}$$

  Monoid: *indexed set union* (a full outer join that unions the groups of matching keys)

- **orderBy** orders key-value pairs by the key:

$$\text{orderBy}(\{(1, a), (2, b), (1, c), (1, d)\}) \ = \ [(1, \{a, c, d\}), (2, \{b\})]$$

  Monoid: merges sorted lists by unioning the groups of matching keys

The Monoid Algebra (cont.)

- **coGroup** is a lossless inner/outer equi-join:

$$\begin{aligned}&\mathrm{coGroup}(\{(1,a),(2,b),(1,c)\},\\&\qquad\{(1,5),(2,6),(3,7)\})\\&\quad=\ \{(1,(\{a,c\},\{5\})),\ (2,(\{b\},\{6\}),\ (3,(\{\},\{7\})))\}\end{aligned}$$

  Monoid: a full outer join that unions groups pairwise

- **reduce** for aggregations:

$$\mathrm{reduce}(+,\{1,2,3\}) = 6$$

  Monoid: $+$

Query Unnesting

- Deriving joins and unnesting any nested query:

$$F(X,Y) \quad = \quad \boxed{\mathrm{flatMap}(\lambda x.\, g(\boxed{\mathrm{flatMap}(\lambda y.\, h(x,y),\ Y)}),X)}$$

$$\rightarrow \quad \mathrm{flatMap}(\lambda(k,(xs,ys)).\, F(xs,ys),\ \mathrm{coGroup}(\mathrm{flatMap}(\lambda x.\, \{(k_1(x),x)\},\ X),\ \mathrm{flatMap}(\lambda y.\, \{(k_2(y),y)\},\ Y)))$$

provided that there are key functions k_1 and k_2 such that
$k_1(x) \neq k_2(y) \ \Rightarrow \ h(x,y) = \{\,\}$

eg, $h(x,y) =$ **if** $k_1(x) == k_2(y)$ **then** e **else** $\{\,\}$

- coGroups are implemented either as distributed-partitioned joins or as broadcast joins

Current Status of DIQL

- is based on compile-time code generation
- uses Scala's compile-time reflection and macros
- is implemented on Spark, Flink, and Cascading/Scalding
- provides a provenance-based debugger

Code Generation

- Run-time code generation (SQL, Hive, Spark SQL, MRQL, . . .)
  - At run-time: validation, optimization, and code generation
  - Can embed values through parametric queries
- Two-stage code generation (DryadLINQ, Emma)
  - At compile-time: validation and static optimizations

    Generates a query graph
  - At run-time: cost-based optimizations and code generation
  - Embedding:
    - DryadLINQ broadcasts embedded values to workers
    - Emma uses run-time reflection to access embedded values
- Compile-time code generation (DIQL)
  - At compile-time: validation, static optimizations, and code generation
  - The optimizer can still do cost-based optimizations:
    1. can pick few viable choices for query plans at compile-time, and
    2. generate conditional code that chooses a plan at run-time