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

Leonidas Fegaras

University of Texas at Arlington

http://lambda.uta.edu/

Outline

- Emerging DISC (Data-Intensive Scalable Computing) systems
- Introducing DIQL: the Data Intensive Query Language
- The monoid algebra
- Incremental query processing

DISC Programming Environments

- Designed for large-scale data processing on computer clusters
- Often work on raw data – indexes are uncommon
- They hide the details of distributed computing, fault tolerance, recovery, etc
- 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
- Hard to develop complex applications when the focus is in optimizing performance
- Too many competing DISC platforms:

  Map-Reduce, Spark, Flink, Storm, ...
- Hard to tell which one will prevail in the near future
- ...or you can use a query language that is independent of the underlying DISC platform!

Limitations of Current DISC Query Languages

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

Most DISC 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
- Example: Hive and DataFrames treat in-memory and distributed collections differently
  - to flatten a nested collection in a row in Spark DataFrames, one must 'explode' the collection
  - Hive supports 'lateral views' to avoid creating intermediate tables when exploding nested collections

Code Generation

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

    Generates a query graph
  - Run-time: cost-based optimizations and code generation
  - Embedding:
    - DryadLINQ broadcasts embedded values to workers
    - Emma uses Scala's run-time reflection to access embedded values
- Compile-time code generation (DIQL)
  - Compile-time: checking, static optimizations, and code generation
  - The optimizer can still do cost-based optimizations:
    1. picks few viable choices for query plans at compile-time
    2. generates conditional code that chooses a plan based on run-time statistics

Wish List for Query Embedding

PL: host programming language, QL: DISC 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 work on distributed collections with special semantics
- Distributed and in-memory collections must be indistinguishable in the QL syntax
  - although they may be processed differently
- No null values in the QL data model
  - SQL uses 3-valued logic (very obscure)
  - most PLs do not provide a standardized way to treat nulls
  - need to handle null values before UDF calls or PL code
- $\Rightarrow$ no outer joins
- nulls in data (unknown/missing values) can be handled with PL code

DIQL (Data-Intensive Query Language)

An SQL-like query language for DISC systems that

- is deeply embedded in Scala
- is optimized and translated to Java byte code at compile-time
- is designed to support multiple Scala-based APIs for DISC processing
  - 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 functional Scala code
  - can use the core Scala libraries and tools, and user-defined classes
- has compositional semantics based on monoid homomorphisms

The DIQL Syntax

$$
\begin{array}{rcll}
\textbf{pattern:} & p & ::= & \textit{any Scala pattern, including a refutable pattern} \\
\\
\textbf{qualifier:} & q & ::= & p <\!\!- e \qquad\qquad \textit{generator over a dataset} \\
 & & | & p <\!\!-\!\!- e \qquad\quad \textit{generator over a small dataset} \\
 & & | & p = e \qquad\qquad\; \textit{binding} \\
\\
\textbf{expression:} & e & ::= & \textit{any Scala functional expression} \\
 & & | & \textbf{select}\ [\,\textbf{distinct}\,]\ e \\
 & & & \qquad \textbf{from}\ q, \ldots, q \\
 & & & \qquad [\,\textbf{where}\ e\,] \\
 & & & \qquad [\,\textbf{group by}\ p\,[\,:e\,]\ [\,\textbf{having}\ e\,]\,] \\
 & & & \qquad [\,\textbf{order by}\ e\,] \\
 & & | & \oplus/e \qquad\qquad \textit{aggregation}
\end{array}
$$

The Group-By Semantics

- Based on comprehensions with ‘order by’ and ‘group by’ [Wadler and Peyton Jones 2007]
- 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 $\{t\}$

  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 $X_{ik} * Y_{kj}$;
  for each group `(i, j)`, the bag `z` contains $X_{ik} * Y_{kj}$, for all k
- `+/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 unnests any nested query to a coGroup

Full Outer Join: Matrix Addition

Matrix addition $X + Y$ is equivalent to a full outer join:

```
select ( +/(x++y), i, j )
from (x,i,j) <- X group by (i,j)
from (y,i_,j_) <- Y group by (i_,j_)
```

- `X` is grouped by `(i,j)`
- `Y` is grouped by `(i_,j_)`
- with `(i,j)==(i_,j_)`

This is a coGroup!

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 DISC 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

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

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

  - Some monoid homomorphisms are not allowed

    can't convert a bag to a list
- Collection monads (aka, ringads) have a monoidal structure too
  - but are not a good basis for practical data-centric languages
  - require various extensions (for group-by, aggregations, etc)

The Monoid Algebra

Generalizes the nested relational algebra (here shown on bags only)

- **flatMap** of type $(\alpha \rightarrow \{\beta\}, \{\alpha\}) \rightarrow \{\beta\}$

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

  Monoid: $\uplus$
- **groupBy** of type $\{(\kappa,\alpha)\} \rightarrow \{(\kappa,\{\alpha\})\}$

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

  It returns an indexed set (aka, a key-value map).
  Monoid: *indexed set union* (a full outer join that unions the groups of matching keys)
- **orderBy** of type $\{(\kappa,\alpha)\} \rightarrow [(\kappa,\{\alpha\})]$
  Monoid: merges sorted lists by unioning the groups of matching keys

The Monoid Algebra (cont.)

- **coGroup** of type $(\{(\kappa,\alpha)\},\ \{(\kappa,\beta)\}) \rightarrow \{(\kappa,(\{\alpha\},\{\beta\}))\}$

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

  A lossless inner/outer equi-join.
  Monoid: a full outer join that unions groups pairwise

- **reduce** of type $((\alpha,\alpha) \rightarrow \alpha,\ \{\alpha\}) \rightarrow \alpha$

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

  Monoid: $+$

The Essence of Data Parallelism

- Divide-and-conquer = groupBy-and-flatMap
- Map-Reduce = flatMap-groupBy-flatMap

$$\mathrm{mapReduce}(m, r)(X) \quad = \quad \mathrm{flatMap}(r, \mathrm{groupBy}(\mathrm{flatMap}(m, X)))$$

for a map function m of type $(k_1, \alpha) \rightarrow \{(k_2, \beta)\}$
and a reduce function r of type $(k_2, \{\beta\}) \rightarrow \{(k_3, \gamma)\}$

Term Normalization

- Fuse two cascaded flatMaps into a nested flatMap:

$$\mathrm{flatMap}(f, \mathrm{flatMap}(g, S)) \rightarrow \mathrm{flatMap}(\lambda x.\, \mathrm{flatMap}(f, g(x)), S)$$

- *Normal form:* a tree of groupBy/coGroup nodes connected via a single flatMap

Query Unnesting

- Deriving joins and unnesting any nested query:

$$
\begin{aligned}
F(X, Y) \quad &= \quad \boxed{\mathrm{flatMap}(\lambda x.\, g(\boxed{\mathrm{flatMap}(\lambda y.\, h(x, y),\ Y)}), X)} \\
&\rightarrow \quad \mathrm{flatMap}(\lambda(k, (xs, ys)).\, F(xs, ys), \\
&\qquad\qquad\qquad \mathrm{coGroup}(\mathrm{flatMap}(\lambda x.\, \{(k_1(x), x)\},\ X), \\
&\qquad\qquad\qquad\qquad\qquad \mathrm{flatMap}(\lambda y.\, \{(k_2(y), y)\},\ Y)))
\end{aligned}
$$

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

- coGroups are implemented as distributed partitioned joins or broadcast joins

Algebraic Terms

- Our monoid algebraic operations are homomorphisms
- ... but flatMap over groupBy or coGroup may not be a homomorphism
  - groupBy returns an indexed set
  - flatMap distributes over $\uplus$ but doesn't distribute over indexed set union
- *Special case:* if g is a homomorphism then so is this:

$$\mathrm{flatMap}(\lambda(k,s).\,\{(k,g(s))\},\ \mathrm{groupBy}(X))$$

  ie, flatMap must propagate the groupBy key
- This is the basis for our incrementalization method

Incremental Stream Processing

Many emerging Distributed Stream Processing Engines (DSPEs)

Storm, Spark Streaming, Flink Streaming

Most are based on *batch streaming*

- continuous processing over streams of batch data
  (data that come in continuous large batches)

We want to:

- convert any batch DISC query to an incremental stream processing programs automatically
- derive incremental programs that return accurate results, not approximate answers — unlike most stream processing systems
  - retain a minimal state during streaming
  - derive an accurate snapshot answer periodically

Why Bother?

Incremental stream processing analyzes data in incremental fashion:

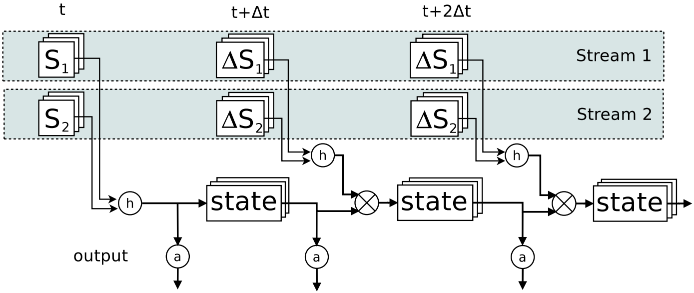
> Existing results on current data are reused and merged with the results of processing new data

Advantages:

- it can achieve better performance and require less memory than batch processing
- it allows to process data streams in real-time with low latency
- it can be used for analyzing very large data incrementally

    in batches that can fit in memory;
    enabling us to process more data with less hardware

Incrementalization using Homomorphisms



A query $q(S_1, S_2)$ over two streams S_1 and S_2 is split into a homomorphism h and an answer function a:

$$q(S_1, S_2) = a(h(S_1, S_2))$$

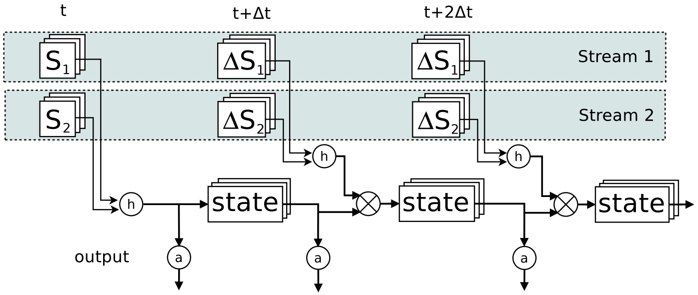
where $h(S_1 \uplus \Delta S_1, S_2 \uplus \Delta S_2) = h(S_1, S_2) \otimes h(\Delta S_1, \Delta S_2)$, for some monoid $\otimes$

Transforming an Algebraic Term to a Homomorphism

- We transform each term to propagate the groupBy and coGroup keys to the output

  known as lineage tracking
- *lineage keys:* the groupBy/coGroup keys in the query
- Why?
  - the query results will be grouped by the lineage keys
    - the current state is kept grouped by the lineage keys
    - the query results on the new data are grouped by the lineage keys
  - the state is combined with the new query results by joining them on the lineage keys using indexed set union

Incremental Processing



$$h(S_1 \uplus \Delta S_1, S_2 \uplus \Delta S_2) = h(S_1, S_2) \otimes h(\Delta S_1, \Delta S_2)$$

- In most cases, the merging $\otimes$ is an indexed set union that joins the current state $h(S_1, S_2)$ with the new results $h(\Delta S_1, \Delta S_2)$
- The state remains partitioned on the lineage keys
- Only $h(\Delta S_1, \Delta S_2)$ needs to be distributed across the workers based on the lineage keys
- We may store the state as a key-value map and update it in place

Current Work

- MRQL: A querying system for Big data analytics
  - run-time code generation using Java reflection
  - implemented on Spark, Flink, Hama, and Storm
- Incremental MRQL is implemented on Spark Streaming
- DIQL: a DISC query system deeply embedded in Scala
  - compile-time code generation using Scala's compile-time reflection and macros
  - implemented on Spark, Flink, and Cascading/Scalding
  - provides a provenance-based debugger

Why not Monads?

- Some DISC frameworks are based on collection monads (ringads)

  ringad = monad + union + zero (a monoidal structure)

- They require various extensions to become a good basis for practical QLs:
  - join heterogeneous collections: need coercions (homomorphisms)
  - group-by a bag: need a group-by homomorphism
  - order-by: need a sorting homomorphism
  - aggregations: need monoid homomorphisms
- There is a pattern here!
- Need monoid homomorphisms, not monads!

  ringads are less expressive than monoids

  ... but have more "structure" than necessary

- Why don't we just extend ringads with a fold?

  bad idea; it may lead to false statements when applied to bags:

  $1 = \text{first}(\{1{,}2\}) = \text{first}(\{2{,}1\}) = 2$, $\text{first}(S) = \text{fold}(\lambda(x,r).\, x)\ 0\ S$

Embedded External DSLs in Scala

- Using existing Scala syntax (eg, for-comprehensions) to simulate DSL syntax is not always a good solution

  internal vs external DSL

- Is there an easy way to extend the Scala syntax with custom DSL syntax?

```
def parse ( parser: => Parser[c.Tree] ): Parser[c.Tree]

def scala: Parser[c.Tree] = parse(scala)

def expr: Parser[c.Tree]
  = ( "select" ~ expr ~ "from" ~ ...
    | parse(expr)
    )
```

- Parser for patterns?