Emma in Action: Declarative Dataflows for Scalable Data Analysis

Alexander Alexandrov  Andreas Salzmann
Georgi Krastev  Asterios Katsifodimos  Volker Markl

TU Berlin, Germany
firstname.lastname@tu-berlin.de

ABSTRACT

Parallel dataflow APIs based on second-order functions were originally seen as a flexible alternative to SQL. Over time, however, their complexity increased due to the number of physical aspects that had to be exposed by the underlying engines in order to facilitate efficient execution. To retain a sufficient level of abstraction and lower the barrier of entry for data scientists, projects like Spark and Flink currently offer domain-specific APIs on top of their parallel collection abstractions.

This demonstration highlights the benefits of an alternative design based on deep language embedding. We showcase Emma – a programming language embedded in Scala. Emma promotes parallel collection processing through native constructs like Scala’s for-comprehensions – a declarative syntax akin to SQL. In addition, Emma also advocates quoting the entire data analysis algorithm rather than its individual dataflow expressions. This allows for decomposing the quoted code into (sequential) control flow and (parallel) dataflow fragments, optimizing the dataflows in context, and transparently offloading them to an engine like Spark or Flink. The proposed design promises increased programmer productivity due to avoiding an impedance mismatch, thereby reducing the lag times and cost of data analysis.

1. INTRODUCTION & MOTIVATION

The last decade was marked by a major shift in data analytics technology. Driven by the need to store and process in-situ data at scale, parallel dataflow engines like Hadoop MapReduce [2], Spark [3] and Flink [1] emerged as an alternative to parallel relational databases. The principle programming abstraction behind these systems is a distributed collection type equipped with second-order functions that encapsulate parallelism. Assembling dataflows by combining these functions offers flexibility, but is hindered by additional complexity as various physical execution aspects of the underlying engines have to be exposed in the API in order to facilitate efficient evaluation. Such issues are: (i) Multi-way joins need to be written as a cascade of binary join function applications. The resulting tree is mirrored at execution time, as no join order optimization is performed at this level; (ii) Parallel aggregates need to be written in a specific way using dedicated primitives like reduceByKey rather than the more general and intuitive groupBy followed by a map; (iii) Since dataflow expressions per default are lazy, the decision to materialize an intermediate result is made explicitly by the programmer with a cache primitive.

We end up in a situation with several well-known problems such as: (i) high barrier of entry due to the required level of understanding of the underlying execution model, (ii) hard to read and maintain code due to low level of abstraction, and (iii) missed opportunities for optimization due to hard-coded execution strategies. In [4] we present a more detailed discussion of the idiosyncratic code patterns resulting from these problems. The principle way to tackle these problems is by providing high-level programming abstractions on top of the low-level dataflow APIs. The merits of this approach are validated by the popularity of external domain-specific languages like Pig, Hive, and SystemML, on the one side, and internal domain-specific libraries like DataFrame APIs, GraphX, MLLib, and Mahout, on the other. Both strategies, however, do not resolve the need for declarative integration of parallel collection processing in a general-purpose language – external languages introduce a language barrier, while internal libraries introduce a domain or API barrier.

This demonstration aims to showcase the benefits of an alternative approach based on deep language embedding through quotation and meta-programming recently proposed in [5]. We show that the ability to manipulate the entire data analysis program at compile time has twofold impact. First, extending ideas explored by LINQ [10] and Ferry [7] to the field of parallel dataflow engines, it facilitates deep linguistic reuse of host language constructs like comprehension syntax for declarative, SQL-like dataflow definitions. Second, it allows for transparently decomposing the program code into a (sequential) driver and multiple (parallel) dataflow fragments. The identified dataflows can then be optimized jointly based on the surrounding driver context before being translated and offloaded to a co-processing parallel dataflow engine like Spark or Flink. The net effect is an high-level collection processing API where notions of parallelism associated with an underlying dataflow engine are hidden from the programmer.
2. FORMAL FOUNDATIONS

Parallel Bags and Folds. We use the theory of recursive data types to model the parallel collection types – Data\textit{Set} in Flink and RDD in Spark – which form the core of the targeted parallel dataflow engines. Both types are oblivious with respect to element order and allow for duplicates and therefore can be accurately modeled by the algebraic data type Bag (as opposed to List which also models order or Set which does not model duplicates).

Depending on the constructor choice, there are two equivalent definitions of the recursive data type Bag\textit{A} usually called insert and union representation \cite{6}. We base our design on the latter, in which bags can be constructed in three ways: by a nullary constructor \textit{emp} denoting the empty bag, an unary constructor \textit{sng} \textit{x} denoting a singleton bag, or a binary constructor \textit{uni} \textit{xs ys} denoting the (duplicate preserving) union of two other bags.

Union representation is better suited for our needs than the more widely used insert representation, as it allows us to characterize parallel collection processing by means of structural recursion (folds) – a method for defining functions on bags \textit{xs} by means of recursive replacement of constructor applications with applications of compatible functions \cite{9}.

```
// structural recursion on union-style bags
def fold[A,B](e: B; s: A => B; u: (B,B) => B)
  (xs: Bag[A]) = xs match {
    case emp         => e
    case sng(x)      => s(x)
    case uni(xs,ys)  => u(fold(e,s,u)(ys), fold(e,s,u)(zs))
  }
```

The \texttt{fold} function takes three parameters: \texttt{e}, \texttt{s}, and \texttt{u}, substitutes them in place of the constructor applications in \texttt{xs}, and evaluates the resulting expression tree to get a final value \( x \in \mathbb{B} \). To compute the sum of all elements, for example, we substitute with \( e = 0 \), \( s = \text{id} \), and \( u = + \):

\[
\begin{array}{c}
\text{uni} \\
\text{uni} \\
\text{uni} \\
\text{sng} \quad \text{sng} \\
\text{sng} \quad \text{sng} \\
\text{emp} \quad \text{id} \\
\text{id} \quad \text{id} \\
\text{id} \quad \text{id} \\
\text{id} \\
\end{array}
\Rightarrow + \\
\Rightarrow + \\
\Rightarrow 15
\]

Aggregations like \texttt{min}, \texttt{max}, \texttt{sum}, and \texttt{count}, existential qualifiers like \texttt{exists} and \texttt{forall}, as well as collection processing operators like \texttt{map} and \texttt{filter} can be defined as folds.

Comprehension Syntax. Using the induced structural recursion scheme of union-style bags, we can define an algebraic structure known as monad on top of Bag\textit{A} and enable declarative dataflow specification.

To illustrate the rationale behind the bag monad, consider two bags \( z = \{\{1,2,2,3\}\} \) and \( y = \{\{1,2\}\} \) with corresponding constructor application trees:

\[
\begin{array}{c}
\text{uni} \\
\text{uni} \\
\text{uni} \\
\text{sng} \quad \text{sng} \\
\text{sng} \quad \text{sng} \\
\text{sng} \quad \text{sng} \\
\text{sng} \quad \text{sng} \\
\text{id} \quad \text{id} \\
\text{id} \quad \text{id} \\
\text{id} \quad \text{id} \\
\text{id} \\
\end{array}
\]

Now consider an example where we want to compute the bag of all pairs \( (x,y) \) where \( x \in z \), \( y \in y \) and \( x = y \).

If \( z \) and \( y \) were sets, we could describe this computation mathematically using set comprehension syntax:

\[
\{ (x, y) | x \in z, y \in y, x = y \}
\]

If \( z \) and \( y \) were relations in a database, we could write a select-from-where query:

```
SELECT x, y FROM z, y AS x WHERE x = y
```

Modern functional languages like Scala allow us to use native comprehension syntax for arbitrary types, as long as those implement the so-called monad operators: map, flatMap, and withFilter. We can then formalize the intended computation as:

```
for (x <- z; y <- y; if x == y) yield (x, y)
```

At parse time, the for-comprehension syntax desugars into a chain of nested flatMap applications ending with a map and interleaved with withFilter. The above code becomes:

```
z.flatMap(x => 
  y.withFilter(y => x == y).map(y => (x, y))
```

How should we interpret this desugaring in terms of the structural recursion scheme discussed above? The map part of the flatMap application operates on the level of the (red) \textit{xs} tree – the tree shape is preserved and each value \( x \) is substituted with a copy of the entire \textit{ys} tree. The inner map operates on the level of the \textit{ys} trees and maps their \( y \) values to a \((x,y)\) pair using the \( x \) from the outer map. We end up with an outer (red) bag of inner (blue) bags.

The withFilter application substitutes singleton bags that do not satisfy the \( x = y \) predicate with emp, and the flat part of flatMap “forgets” the nested bag structure by inlining the inner trees into the outer one.

Theory to Practice. Bag comprehensions provide the key ingredient for solving two long-standing problems. First, as first-class citizen in a general-purpose source language, comprehension syntax offers direct means for declarative parallel collection processing that can be seen as a generalization of SQL. Second, as first-class citizen in an object language that can be manipulated through meta-programming, comprehensions can serve as an entry point for the integration of
3. EMMA: MAIN CONCEPTS & DESIGN

Based on the formal foundations outlined in Section 2, in [5] we proposed Emma—a declarative API for parallel collection processing deeply embedded in Scala. The core abstraction exposed by Emma is a generic type DataBag which models bags in union representation. In the following, we present the most distinguishing features by example.

Declarative SPJ Expressions. Binary operators like join and cross are not included in the API. Instead, the DataBag type implements the monad operators discussed in Section 2. This allows for writing dataflows in a declarative syntax similar to the Select-Project-Join style known from SQL. To illustrate this, consider the following code snippet taken from an Emma implementation of the Alternating Least Squares (ALS) algorithm as described by [11].

```scala
val prefs = for {
  user <- users
  rating <- ratings
  item <- items
  if rating.user == user.id
  if rating.item == item.id
} yield Preference(user, item.F, rating.value)
```

Multi-way joins like the example above are identiﬁed by the Emma compiler and translated as join cascades in the backend engine.

Nesting. The groupBy operator adds a level of nesting:

```scala
val ys: DataBag[Group[K, DataBag[A]]] = xs.groupBy(k)
```

3. EMMA: MAIN CONCEPTS & DESIGN

Based on the formal foundations outlined in Section 2, in [5] we proposed Emma—a declarative API for parallel collection processing deeply embedded in Scala. The core abstraction exposed by Emma is a generic type DataBag which models bags in union representation. In the following, we present the most distinguishing features by example.

Declarative SPJ Expressions. Binary operators like join and cross are not included in the API. Instead, the DataBag type implements the monad operators discussed in Section 2. This allows for writing dataflows in a declarative syntax similar to the Select-Project-Join style known from SQL. To illustrate this, consider the following code snippet taken from an Emma implementation of the Alternating Least Squares (ALS) algorithm as described by [11].

```scala
val prefs = for {
  user <- users
  rating <- ratings
  item <- items
  if rating.user == user.id
  if rating.item == item.id
} yield Preference(user, item.F, rating.value)
```

Multi-way joins like the example above are identiﬁed by the Emma compiler and translated as join cascades in the backend engine.

Nesting. The groupBy operator adds a level of nesting:

```scala
val ys: DataBag[Group[K, DataBag[A]]] = xs.groupBy(k)
```

3. EMMA: MAIN CONCEPTS & DESIGN

Based on the formal foundations outlined in Section 2, in [5] we proposed Emma—a declarative API for parallel collection processing deeply embedded in Scala. The core abstraction exposed by Emma is a generic type DataBag which models bags in union representation. In the following, we present the most distinguishing features by example.

Declarative SPJ Expressions. Binary operators like join and cross are not included in the API. Instead, the DataBag type implements the monad operators discussed in Section 2. This allows for writing dataflows in a declarative syntax similar to the Select-Project-Join style known from SQL. To illustrate this, consider the following code snippet taken from an Emma implementation of the Alternating Least Squares (ALS) algorithm as described by [11].

```scala
val prefs = for {
  user <- users
  rating <- ratings
  item <- items
  if rating.user == user.id
  if rating.item == item.id
} yield Preference(user, item.F, rating.value)
```

Multi-way joins like the example above are identiﬁed by the Emma compiler and translated as join cascades in the backend engine.

Nesting. The groupBy operator adds a level of nesting:

```scala
val ys: DataBag[Group[K, DataBag[A]]] = xs.groupBy(k)
```

3. EMMA: MAIN CONCEPTS & DESIGN

Based on the formal foundations outlined in Section 2, in [5] we proposed Emma—a declarative API for parallel collection processing deeply embedded in Scala. The core abstraction exposed by Emma is a generic type DataBag which models bags in union representation. In the following, we present the most distinguishing features by example.

Declarative SPJ Expressions. Binary operators like join and cross are not included in the API. Instead, the DataBag type implements the monad operators discussed in Section 2. This allows for writing dataflows in a declarative syntax similar to the Select-Project-Join style known from SQL. To illustrate this, consider the following code snippet taken from an Emma implementation of the Alternating Least Squares (ALS) algorithm as described by [11].

```scala
val prefs = for {
  user <- users
  rating <- ratings
  item <- items
  if rating.user == user.id
  if rating.item == item.id
} yield Preference(user, item.F, rating.value)
```

Multi-way joins like the example above are identiﬁed by the Emma compiler and translated as join cascades in the backend engine.

Nesting. The groupBy operator adds a level of nesting:

```scala
val ys: DataBag[Group[K, DataBag[A]]] = xs.groupBy(k)
```

3. EMMA: MAIN CONCEPTS & DESIGN

Based on the formal foundations outlined in Section 2, in [5] we proposed Emma—a declarative API for parallel collection processing deeply embedded in Scala. The core abstraction exposed by Emma is a generic type DataBag which models bags in union representation. In the following, we present the most distinguishing features by example.

Declarative SPJ Expressions. Binary operators like join and cross are not included in the API. Instead, the DataBag type implements the monad operators discussed in Section 2. This allows for writing dataflows in a declarative syntax similar to the Select-Project-Join style known from SQL. To illustrate this, consider the following code snippet taken from an Emma implementation of the Alternating Least Squares (ALS) algorithm as described by [11].

```scala
val prefs = for {
  user <- users
  rating <- ratings
  item <- items
  if rating.user == user.id
  if rating.item == item.id
} yield Preference(user, item.F, rating.value)
```

Multi-way joins like the example above are identiﬁed by the Emma compiler and translated as join cascades in the backend engine.

Nesting. The groupBy operator adds a level of nesting:

```scala
val ys: DataBag[Group[K, DataBag[A]]] = xs.groupBy(k)
```

3. EMMA: MAIN CONCEPTS & DESIGN

Based on the formal foundations outlined in Section 2, in [5] we proposed Emma—a declarative API for parallel collection processing deeply embedded in Scala. The core abstraction exposed by Emma is a generic type DataBag which models bags in union representation. In the following, we present the most distinguishing features by example.

Declarative SPJ Expressions. Binary operators like join and cross are not included in the API. Instead, the DataBag type implements the monad operators discussed in Section 2. This allows for writing dataflows in a declarative syntax similar to the Select-Project-Join style known from SQL. To illustrate this, consider the following code snippet taken from an Emma implementation of the Alternating Least Squares (ALS) algorithm as described by [11].

```scala
val prefs = for {
  user <- users
  rating <- ratings
  item <- items
  if rating.user == user.id
  if rating.item == item.id
} yield Preference(user, item.F, rating.value)
```

Multi-way joins like the example above are identiﬁed by the Emma compiler and translated as join cascades in the backend engine.

Nesting. The groupBy operator adds a level of nesting:

```scala
val ys: DataBag[Group[K, DataBag[A]]] = xs.groupBy(k)
```
Figure 2: Two stages in the execution of an Alternating Least Squares algorithm in Emma.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type/Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>K-Means</td>
<td>Clustering</td>
</tr>
<tr>
<td>Naive Bayes</td>
<td>Classification</td>
</tr>
<tr>
<td>LabelRank</td>
<td>Semi-supervised classification</td>
</tr>
<tr>
<td>Alternating Least Squares</td>
<td>Collaborative filtering</td>
</tr>
<tr>
<td>Belief Propagation</td>
<td>Statistical inference</td>
</tr>
<tr>
<td>Graph Colouring</td>
<td>Graph analysis</td>
</tr>
<tr>
<td>Triangle Count</td>
<td>Graph analysis</td>
</tr>
<tr>
<td>Tie-Tac-Toe</td>
<td>Retrograde analysis</td>
</tr>
<tr>
<td>TPC-H Queries</td>
<td>Relational</td>
</tr>
</tbody>
</table>

Table 1: Example algorithms implemented in Emma.

1. On the left pane, audience members can select between several data-analysis algorithms written in Emma and review their source code (Table 1).

2. The algorithm code is quoted with a `parallelize` macro and transformed into an Algorithm object that can be evaluated by an Emma-supported runtime – we offer a choice between Spark or Flink. Execution is triggered by pressing a “Play” button from the GUI.

3. The GUI pauses before submitting the first dataflow, marks the corresponding DataBag term in the left pane, and shows the dataflow graph compiled by the Emma JIT component on the right pane. Optimizations performed by Emma are depicted on the visualized graph.

4. Upon renewed pressing of the “Play” button, the current dataflow and the subsequent driver expressions are evaluated. Execution halts at each subsequent dataflow in a manner similar to step 3.

5. A graph that marks adjacent dataflows is lazily constructed and visualized. The cycles and branches in the graph reflect the control-flow structure in the part of the code which constitutes the algorithm driver.

Finally, in Part 3 we invite the audience to try out the Emma API and define other data analysis algorithms. The produced programs can be executed interactively from the GUI just as the predefined examples from Part 2.

5. ACKNOWLEDGEMENTS

This work has been supported through grants by the German Science Foundation (DFG) as FOR 1306 Stratosphere, by the German Ministry for Education and Research as Berlin Big Data Center BBDC (ref. 01IS14013A), by the European Commission as Proteus (ref. 687691) and Streamline (ref. 688191), and by Oracle Labs.

6. REFERENCES