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 a high-level collection processing API where notions of parallelism associated with an underlying dataflow engine are hidden from the programmer.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2899396

2. FORMAL FOUNDATIONS

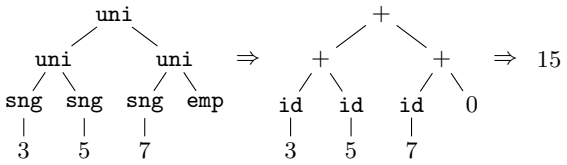
Parallel Bags and Folds. We use the theory of recursive data types to model the parallel collection types – `DataSet` in Flink and `RDD` in Spark – which form the core of the targeted parallel dataflow engines. Both types are oblivious with respect to element order and allow for duplicates and therefore 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 A usually called *insert* and *union* representation [6]. We base our design on the latter, in which bags can be constructed in three ways: by a nullary constructor `emp` denoting the empty bag, an unary constructor `sng` x denoting a singleton bag, or a binary constructor `uni` 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 xs by means of recursive replacement of constructor applications with applications of compatible functions [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(ys,zs) => u(fold(e,s,u)(ys), fold(e,s,u)(zs))
}
```

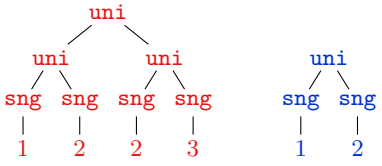
The `fold` function takes three parameters: `e`, `s`, and `u`, substitutes them in place of the constructor applications in xs, and evaluates the resulting expression tree to get a final value $z \in$ B. To compute the sum of all elements, for example, we substitute with `e` $= 0$, `s` $=$ `id`, and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



Aggregations like `min`, `max`, `sum`, and `count`, existential qualifiers like `exists` and `forall`, as well as collection processing operators like `map` and `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 A and enable declarative dataflow specification.

To illustrate the rationale behind the bag monad, consider two bags $xs = \{\{1,2,2,3\}\}$ and $ys = \{\{1,2\}\}$ with corresponding constructor application trees:



Now consider an example where we want to compute the bag of all pairs (x, y) where $x \in xs$, $y \in ys$ and $x = y$. If xs and ys were sets, we could describe this computation mathematically using set comprehension syntax:

$$\{\, (x, y) \mid x \in xs,\ y \in ys,\ x = y \,\}$$

If xs and ys were relations in a database, we could write a select-from-where query:

```
SELECT x, y FROM xs as x, ys as y WHERE x = y
```

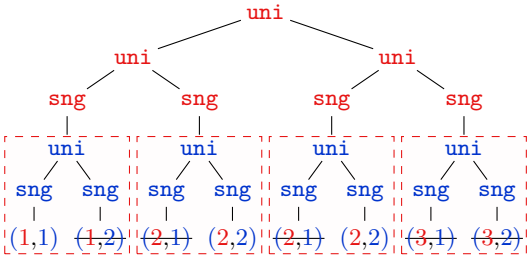
Modern functional languages like Scala allow us to use native comprehension syntax for arbitrary types, as long as those implement the so-called monad operators: `map`, `flatMap`, and `withFilter`. We can then formalize the intended computation as:

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

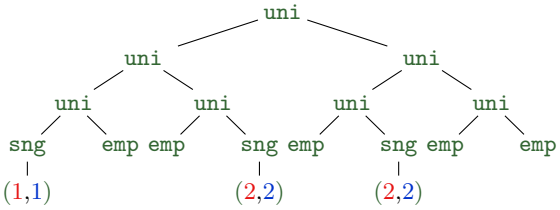
At parse time, the `for`-comprehension syntax desugars into a chain of nested `flatMap` applications ending with a `map` and interleaved with `withFilter`. The above code becomes:

```
xs.flatMap(x =>
  ys.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) xs tree – the tree shape is preserved and each value x is substituted with a copy of the entire ys tree. The inner `map` operates on the level of the 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 trough meta-programming, comprehensions can serve as an entry point for the integration of

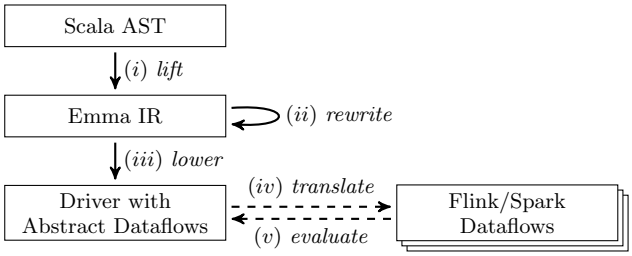


Figure 1: Emma compiler pipeline.

database optimization techniques into general-purpose languages. This allows for avoiding the naïve evaluation semantics by rewriting comprehended terms as join cascades [8].

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]. The code computes a collection of preference objects that associate users with the feature vectors and the ratings of their related (i.e. rated) items.

```
val prefs = for {
  user   <- users
  rating <- ratings
  item   <- items
  if rating.user == user.id
  if rating.item == item.id
} yield Pref(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 API of the backend engine.

Folds. Computing information from the contents of a `DataBag` is allowed only by means of structural recursion. To that end, we expose the `fold` operator from Section 2 as well as pre-defined aliases for commonly used folds (e.g. `count`, `exists`, `minBy`). Counting the number of users, for example, can be written as:

```
val N = users.fold(0, x => 1, (x, y) => x + y) // or
val N = users.count() // alias for the above
```

Nesting. The `groupBy` operator adds a level of nesting:

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

The resulting bag contains groups of `values` that share the same `key` and `values` which again have type `DataBag[A]`. This is different from Spark and Flink, where the type of the group values is either `Iterable[A]` or `Iterator[A]` and nesting is not natively supported at the API level. The ability to nest `DataBag` instances allows for hiding the complexity of primitives like `groupByKey`, `reduceByKey`, and `aggregateByKey` behind an ubiquitous API in which bags can be processed in a nested manner. Continuing with the next step in the ALS example, the code that updates the user models from the preferences can be written as follows.

```
users = for { // update users
  (user, prefs) <- prefs.groupBy(p => p.user)
} yield { // calculate new feature vector per user
  val Vu = prefs.fold(V0)(p => p.F * p.rating, _ + _)
  val Au = prefs.fold(M0)(p => p.F outer p.F, _ + _)
  val Eu = E * (lambda * user.id)
  user.copy(F = inv(Au + Eu) * Vu)
}
```

As before, due to the fact that the whole expression can be manipulated at compile time, we can recognize nested `DataBag` patterns like the one above and rewrite them using more efficient primitives like `aggregateByKey`.

Coarse-Grained Parallelism Contracts. Dataflow APIs currently provide data-parallelism contracts at the operator level (e.g. `map` for element-at-a-time, `join` for pair-at-a-time, etc.). Emma takes a different approach as it's `DataBag` abstraction itself serves as a *coarse-grained* contract for data-parallel computation. The promise Emma gives to its users is to (*i*) discover all maximal `DataBag` expressions in a quoted code fragment, (*ii*) rewrite them logically in order to maximize the degree of data-parallelism, (*iii*) take a holistic approach while translating them as parallel dataflow expressions, and (*iv*) transparently place primitives that influence physical execution aspects like `broadcast` and `cache`.

Compiler Pipeline. Figure 1 depicts Emma's compiler pipeline. Scala expressions are quoted in a `parallelize` macro which exposes the Abstract Syntax Tree (AST) of the quoted code to the Emma compiler. The compiler first lifts the AST to a suitable intermediate representation (IR), and then performs joint logical and physical rewrites (e.g. inlining, fold-group fusion) on the identified dataflow fragments. The rewritten IR is finally lowered and compiled as a driver with abstract dataflow expressions. At runtime, these dataflow expressions are translated Just in Time (JIT) and evaluated on a target dataflow engine (Flink or Spark).

4. DEMONSTRATION

The demonstration is structured in three parts and showcases the main features of Emma highlighted in Sections 2 and 3 by example.

In Part ❶, we offer a gentle introduction to monads and introduce Emma's `DataBag` API. We demonstrate (*i*) the ability to use comprehension syntax instead of `join` cascades, (*ii*) the support for nesting, and (*iii*) `fold` as the main primitive for parallel computation. We compare the expressiveness offered by Emma against the parallel dataflow APIs targeted as a backend, SQL (as a de-facto standard for declarativity), as well as against domain-specific APIs offered by the backend engines (e.g. DataFrames in Spark, Table API in Flink).

In Part ❷, we showcase the compilation pipeline described in Section 3 and the core principles advocated by our design – coarse-grained parallelism contracts and coarse-grained quotation. To that end, we provide a graphical user interface (GUI) with two panes (Figure 2).

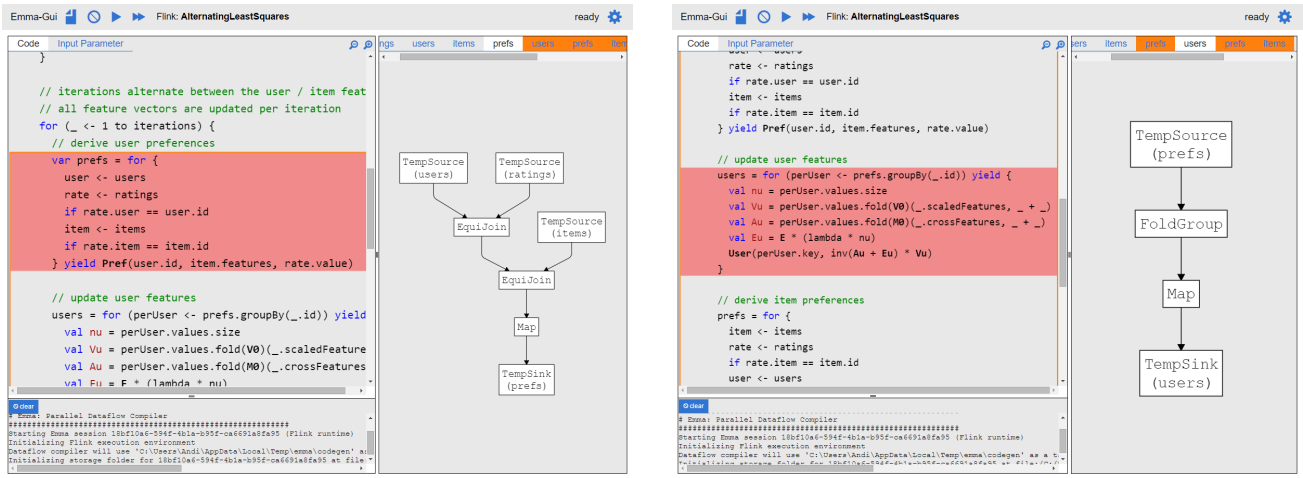


Figure 2: Two stages in the execution of an Alternating Least Squares algorithm in Emma.

Algorithm	Type/Domain
K-Means	Clustering
Naive Bayes	Classification
LabelRank	Semi-supervised classification
Alternating Least Squares	Collaborative filtering
Belief Propagation	Statistical inference
Graph Colouring	Graph analysis
Triangle Count	Graph analysis
Tic-Tac-Toe	Retrograde analysis
TPC-H Queries	Relational

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 ❸ 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 ❷.

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

[1] Apache Flink Project. http://flink.apache.org.
[2] Apache Hadoop Project. http://hadoop.apache.org.
[3] Apache Spark Project. http://spark.apache.org.
[4] A. Alexandrov, A. Katsifodimos, G. Krastev, and V. Markl. Implicit parallelism through deep language embedding. *SIGMOD Record*, 2016 (to appear).
[5] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit parallelism through deep language embedding. In *SIGMOD Conference*, pages 47–61. ACM, 2015.
[6] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 149(1):3–48, 1995.
[7] G. Giorgidze, T. Grust, T. Schreiber, and J. Weijers. Haskell boards the ferry - database-supported program execution for Haskell. In *IFL*, volume 6647 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.
[8] T. Grust and M. H. Scholl. How to comprehend queries functionally. *J. Intell. Inf. Syst.*, 12(2-3):191–218, 1999.
[9] G. L. S. Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *ICFP*, pages 1–2. ACM, 2009.
[10] E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD Conference*, page 706. ACM, 2006.
[11] Y. Zhou, D. M. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix Prize. In *AAIM*, volume 5034 of *Lecture Notes in Computer Science*, pages 337–348. Springer, 2008.