Implicit Parallelism through Deep Language Embedding

Alexander Alexandrov Asterios Katsifodimos Georgi Krastev
Volker Markl

TU Berlin
first.last@tu-berlin.de

ABSTRACT

Parallel collection processing based on second-order functions such as `map` and `reduce` has been widely adopted for scalable data analysis. Initially popularized by Google, over the past decade this programming paradigm has found its way in the core APIs of parallel dataflow engines such as Hadoop's MapReduce, Spark's RDDs, and Flink's DataSets. We review programming patterns typical of these APIs and discuss how they relate to the underlying parallel execution model. We argue that fixing the abstraction leaks exposed by these patterns will reduce the cost of data analysis due to improved programmer productivity. To achieve that, we first revisit the algebraic foundations of parallel collection processing. Based on that, we propose a simplified API that (*i*) provides proper support for nested collection processing and (*ii*) alleviates the need of certain second-order primitives through comprehensions – a declarative syntax akin to SQL. Finally, we present a metaprogramming pipeline that performs algebraic rewrites and physical optimizations which allow us to target parallel dataflow engines like Spark and Flink with competitive performance.

1. INTRODUCTION

One can argue that the success of Google's MapReduce programming model [5] is largely due to its expressiveness and simplicity. Exposing an API built around second-order functions such as `map` f and `reduce` h enables general-purpose programming with collections via user-defined functions (UDFs) f and h. At the same time, the semantics of `map` and `reduce` alone (i.e., regardless of their UDF parameters) enable data-parallelism and facilitate scalability.

Vanilla MapReduce is a perfect fit for generalized processing and aggregation of a single collection of complex objects, but programmers stretch its limits when trying to express algorithms characterized by multiple inputs and non-trivial data and control flow dependencies. To overcome these limitations without sacrificing the benefits of seamless integration of UDFs and collection processing primitives in a general-purpose host language like Java or Scala, projects like Cascading [1], SCOPE [9], Spark [23], and Stratosphere/Flink [3] emerged, proposing programming model extensions that focus on two basic aspects:

- **Expressive dataflows APIs**. This includes more second-order primitives (e.g., `cogroup`, `cross`, `join`) as well as means to construct advanced dataflows by composing them freely.
- **Non-trivial data and control flow**. This includes primitives for data exchange between the driver and the UDFs, caching, as well as primitives that enable native control flow.

The original version of this article was published in SIGMOD 2015.

To motivate the goal of this work, we revisit some programming patterns associated with the features listed above. The code examples are given in Spark's Resilient Distributed Datasets (RDD) API, although similar observations can be made in the other APIs as well. The domain used in the examples consists of the following Scala types:

```
case class Person(id: Long, email: String, name: String)

case class Email(id: Long, from: String,
                 to: String, msg: String)
```

Join Cascades. In our first example, we want to write code that associates all emails with their sender and receiver.

```
// 1) join 'people' with 'emails' on 'sender'
val xs = people.map(p => (p.email, p)) join
         emails.map(e => (e.from, e))
// 2) join 'people' with 'xs' on 'receiver'
val ys = people.map(p => (p.email, p)) join
         xs.map(x => (x._2._2.to, x._2))
// 3) transform 'ys' as an RDD of flat tuples
val rs = ys.map(y => {
  val to    = y._2._1     // project 'to'
  val from  = y._2._2._1  // project 'from'
  val email = y._2._2._2  // project 'email'
  (from, to, email)
})
```

Two problems become evident from the above code snippet. First, since Spark models keys as data, inputs on key-based operators such as `join` need to be explicitly transformed into RDDs of `(key, value)` pairs, which enforces an extra `map` before each `join`. Second, n-way joins must be specified as a cascade of binary joins. The element type in the end-result (`ys`) is therefore a tuple of nested pairs whose shape reflects the tree shape of the join cascade. Accessing base data requires projection chains that traverse the tuple tree to its leafs. As the example above illustrates, such idiomatic programming patterns lead to cluttered and hard to read code. Flink and Scalding manage to resolve the first issue by modeling keys as functions rather than data. The second is usually alleviated through weaker schema models, like the field literal lists used in Cascading and Scalding.

Aggregates. For our second example, consider a situation where we want to compute a tf-idf statistic over the collection of emails. We start by calculating the term frequencies – we tokenize each email message into terms `t` and compute their frequencies `tfrq` using a library function, extend the resulting `(t, tfrq)` sequence with the enclosing email identifier, and finally flatten the result.

```
val tf = emails.flatMap(email => {
  tokenizeAndCount(email.msg).map {
    case (t, tfrq) => Tf(email.id, t, tfrq)
  }
})
```

Next, we have to calculate the inverse document frequencies, which for a term t and a document corpus D with $|D| = N$ are given by the following formula.

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{\, d \in D \mid t \in d \,\}|} \quad .$$

One way to do this in Spark is to group by term and map over the groups to calculate the idf values.

```
val N = emails.count().toDouble
val idf = tf
  .groupBy { case (_, t, _) => t }
  .map { case(t, docs) => (t, math.log(N / docs.size)) }
```

However, the proper way to express this computation is with a `reduceByKey` followed by a `map`.

```
val idf = tf
  .map { case (_, t, _) => (t, 1) }
  .reduceByKey(_ + _)
  .map { case (t, dfrq) => (t, math.log(N / dfrq)) }
```

Although they denote the same result, the two variants define different computations. The first shuffles the entire input and materializes the groups at the receiver side, whereas the second computes partial aggregates locally before shuffling and merging the final document frequencies `dfrq`. The difference in performance gets even more dramatic for naturally occurring skewed distributions, as in this case above where the terms follow a Zipf distribution. Programmers are therefore advised to use the `reduceByKey` pattern whenever possible. As in the previous case, this leads to situations where the code is organized according to execution model specifics rather than readability. Again, the problem can be witnessed across all similar APIs (Flink, Cascading, etc.).

Caching. To motivate the need for caching, consider the following piece of code which computes the `tfidf` values out of `tf` and `idf` and uses them to iteratively update a model:

```
val tfidf: RDD[TfIdf] = /* compute 'tf*idf' per term */
var model: RDD[Model] = /* initialize an ML model */
while (...) { // update 'model' until convergence
  model = /* derive from 'tfidf' and old 'model' */
}
```

The problem with that code is that `RDD` expressions are lazy. Under the hood, the `RDD` type implements a builder pattern that accumulates calls of *transformation* primitives like `map`, `flatMap`, `groupBy` into a parallel execution plan. Evaluation is implicitly forced by *action* primitives like `count`, `collect`, `first`. The implication for the code above is that the `tfidf` term is evaluated once for every loop. To fix this type of problems, Spark offers a `cache` primitive that forces an `RDD` result to be persisted. In our example, we can append `cache` to the definition of `tfidf`. As before, figuring out when to use it is left to the programmer and requires understanding and consideration of Spark's execution model. Cascading does not offer explicit support for caching, while Flink can infer and enforce it, but only in limited situations (as we will discuss shortly).

Broadcast Variables. Another problem can arise in a variation of the last example:

```
val tfidf: Seq[TfIdf] = /* compute as above */.collect()
var model: RDD[Model] = /* initialize an ML model */
while (...) { // update 'model' until convergence
  model = /* transform old 'model' */.map(x => {
    /* anonymous function that reads 'tfidf' */
  })
}
```

Evaluation of `tfidf` is now triggered outside of the loop by the `collect` action. The result of the computation is collected as a local sequence and subsequently read in the `map` UDF. As part of its closure, the `tfidf` value is therefore serialized and shipped to the cluster in each iteration. In this situation, performance can be improved by wrapping the `tfidf` definition in a `sc.broadcast(...)` call. The value is then broadcast to the cluster only once outside of the loop and can be subsequently accessed from the `map` UDF through a `tfidf.value` call. As in the previous case, the decision whether to ship a read-only value as part of the UDF closure or as a broadcast variable is left to the programmer. A similar construct exists in Flink's API.

Control Flow. The final issue we highlight is concerned with the form of certain control flow primitives. In order to optimally express the while loop from the caching example in Flink, for example, one has to phrase the program as follows:

```
val tfidf: DataSet[TfIdf] = /* compute per term */
var model: DataSet[Model] = /* initialize an ML model */
model = model.iterate(model => {
  /* transform 'tfidf' and old 'model' */
})
```

Note how Flink relies on a dedicated `iterate` construct for the iterative part of the program. The reason for this is once more tied to the underlying execution mode. Spark supports only acyclic dataflows and realizes iterative computation by lazily unrolling and evaluating dataflows from a Scala-driven loop. Flink's runtime, in contrast, offers restricted support for native iterations. This approach has performance benefits (e.g. less scheduling overhead, loop-invariant data caching), but requires special feedback edges in the dataflow graph. To introduce those edges, a dedicated construct like `iterate` is required at the API level. Ideally, however, this should be hidden from the programmer, who should be able to use a native Scala `while` loop in both cases.

Problem Statement. The above examples highlight the existence of specific programming patterns and primitives across various parallel dataflow APIs and execution engines. The common theme that shines through is the tight interplay between the programming interface and the underlying execution model. We end up in a situation with several well-known problems: (*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 abstractions, and (*iii*) missed opportunities for optimization due to hard-coded execution strategies.

One way to tackle these problems is to provide high-level programming abstractions on top of the low-level dataflow APIs. The merits of this strategy are validated by the popularity of external languages such as Pig, Hive, and SystemML, on the one side, and specialized internal libraries for relational processing (DataFrame APIs), graph analysis (GraphX, GraphLab), and machine learning (MLLib, Mahout), on the other. This development, however, does not resolve the need for seamless, high-level integration of parallel collection processing in a general-purpose language. External languages introduce a language barrier, while internal libraries introduce a domain barrier.

In this paper, we argue in favor of an alternative approach based on *deep language embedding* through quotation and metaprogramming. The ability to manipulate data analysis programs at compile time has twofold impact. First, it facilitates declarative, SQL-like dataflow definitions through host language constructs such as comprehensions. Second, it allows to decompose the program code as combination of (parallel) dataflow and (sequential) driver fragments, and make holistic decisions for optimal dataflows execution based on the surrounding driver context. The overall effect is a high-level collection processing API where notions of parallelism associated with an underlying dataflow engine are hidden from the programmer.

The remainder of this paper is structured as follows. Section 2 reviews a theoretical model for parallel collection processing and its relation to comprehensions – a declarative syntax generalizing SQL. Section 3 presents Emma [2] – a domain-specific language (DSL) embedded in Scala which enables parallel collection processing through comprehensions. Section 4 sketches Emma's compiler pipeline and discusses how traditional database optimizations such as partial aggregates and join order can be revised in light of the model from Section 2. Section 5 reviews related work, and Section 6 discusses ideas for future research. For a more detailed technical description, we refer the reader to the original version of this paper [4].

2. FORMAL FOUNDATIONS

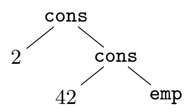
Spark's `RDD`, Cascading's `Collection`, and Flink's `DataSet` all represent homogeneous distributed collections with so called "bag semantics". That is, the elements in a bag share the same type, their order is not fixed, and duplicates are allowed. Our point of departure therefore is a suitable formal model for distributed collections (Section 2.1) and parallel computations on those (Section 2.2), which also facilitates declarative expression syntax (Section 2.3).

2.1 Bags as Algebraic Data Types

Bag Structure. We define the polymorphic type $\mathsf{Bag}\,A$ structurally, using a *recursive constructor algebra.*

$$\mathsf{type}\ \mathsf{Bag}\,A \ = \ \mathtt{emp} \mid \mathtt{cons}\,x\colon A\ xs\colon \mathsf{Bag}\,A \qquad (\text{AlgBag-Ins})$$

The above definition states that all possible $\mathsf{Bag}\,A$ values can be constructed inductively by two primitive constructor functions: `emp` (which denotes the empty bag), and `cons` (which denotes the bag where the element x is added to the bag xs). In other words, for each $xs \in \mathsf{Bag}\,A$ there is a corresponding functional expression t_{xs} (which we call the *constructor application tree*) that constructs xs. For example, the tree associated with $\{\{\,2, 42\,\}\}$ looks as follows.



Bag Semantics. By definition, constructor algebras like AlgBag-Ins are *initial* and thereby, following Lambek's lemma [17], *bijective.* This means that the association between constructor application trees and values is bidirectional – each constructor application tree t_{xs} represents precisely one bag xs and vice versa. This poses a problem, as it contradicts our intended semantics, which state that element order should does not matter. Using only the algebra definition, we have $\{\{\,2, 42\,\}\} \neq \{\{\,42, 2\,\}\}$ because the corresponding trees are different. To overcome this problem, we must add an appropriate *semantic equation.*

$$\mathtt{cons}\,x_1\,\mathtt{cons}\,x_2\ xs = \mathtt{cons}\,x_2\,\mathtt{cons}\,x_1\ xs \quad (\text{EQ-Comm-Ins})$$

The equation states that the order of element insertion is irrelevant for the constructed value. Based on this equation, we can create an equivalence relation on trees and use the induced equivalence classes $[t_{xs}]$ instead of the original trees to ensure $xs \leftrightarrow [t_{xs}]$ bijectivity. In our running example, substituting the trees for $\{\{\,2, 42\,\}\}$ and $\{\{\,42, 2\,\}\}$ in the left- and right-hand sides of (EQ-Comm-Ins), correspondingly, renders them equivalent and puts them in the same equivalence class $[\,\{\{\,2, 42\,\}\}\,]$.

Relevance for Data Management. Conceptually, the $xs \mapsto t_{xs}$ direction can be interpreted as a recursive parser that decomposes a bag xs into its constituting elements. Database runtimes encapsulate this behavior in a reusable operator called `Scan`, and use it to sequentially read the records in a base table. Indeed, we can define a simple iterator-based version of `Scan` with the help of the AlgBag-Ins constructors (again using Scala syntax).

```
class Scan(var xs: Bag[A]) {
  def next(): Option[A] = xs match {
    case emp         => Option.empty[A]
    case cons(x, ys) => xs = ys; Some(x)
  }
}
```

Union Representation. The constructors in AlgBag-Ins impose a *left-deep* structure on the constructor application trees. There is, however, another algebra and a corresponding set of semantic equations that encodes the same initial semantics by means of *general binary trees.*

$$\begin{aligned} \mathsf{type}\ \mathsf{Bag}\,A \ = \ & \mathtt{emp} \\ \mid\ & \mathtt{sng}\,x\colon A \qquad\qquad (\text{AlgBag-Union}) \\ \mid\ & \mathtt{uni}\,xs\colon \mathsf{Bag}\,A\ ys\colon \mathsf{Bag}\,A \end{aligned}$$

$$\begin{aligned} \mathtt{uni}\,xs\,\mathtt{emp} &= \mathtt{uni}\,\mathtt{emp}\,xs \ = \ xs & (\text{EQ-Unit}) \\ \mathtt{uni}\,xs\ (\mathtt{uni}\,ys\,zs) &= \mathtt{uni}\ (\mathtt{uni}\,xs\,ys)\,zs & (\text{EQ-Assoc}) \\ \mathtt{uni}\,xs\,ys &= \mathtt{uni}\,ys\,xs & (\text{EQ-Comm}) \end{aligned}$$

Here `emp` denotes the empty bag, `sng` x denotes the singleton bag, and `uni` $xs\,ys$ denotes the union of xs and ys.

Although AlgBag-Ins and AlgBag-Union model the same type semantics, the latter reflects better the essence of parallel collection processing, as we will show in Section 2.2.

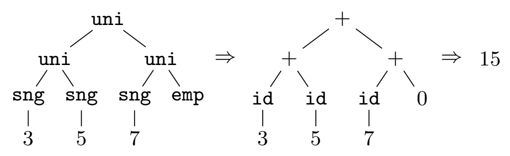
2.2 Structural Recursion on Bags

The previous section described a conceptual model for the structure of bags that identifies bag values with equivalence classes of constructor application trees. We now describe the principle of *structural recursion* – a method for defining functions on bags xs by means of systematic substitution of the constructor applications in the associated t_{xs} tree.

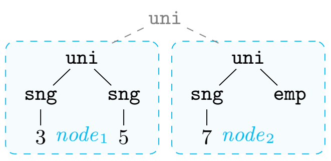
Basic Principle. Consider a case where we want to compute the sum of the elements in $xs = \{\{3, 5, 7\}\}$. We can define this operation with a higher-order function called `fold`.

```
// 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))
}
```

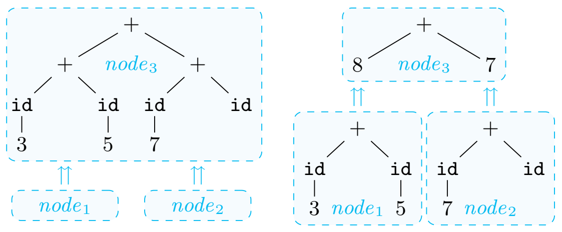
The `fold` function takes three function arguments: `e`, `s`, and `u`, substitutes them in place of the constructor applications in t_{xs}, and evaluates the resulting expression tree to get a final value $z \in \mathsf{B}$. To compute the sum of all elements, for example, we can substitute `e` $= 0$,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



Relevance for Parallel Data Management. Again, we want to highlight the importance of this view on bag computations from a data management perspective. Imagine a scenario where xs is partitioned and distributed over two nodes: $xs_1 = \{\{3, 5\}\}$ and $xs_2 = \{\{7\}\}$. Conceptually, the value is still $xs =$ `uni` xs_1 xs_2, but the `uni` is evaluated only if we have to materialize xs in a single node.



If we need the xs only to apply a `fold`, we can push the `fold` argument functions to the nodes containing xs_i, apply the `fold` locally, and ship the computed z_i values instead. In general, `e`, `s`, and `u` do not form an initial algebra. This implies loss of information when the substituted t_{xs} tree is evaluated to z, and thereby that z is "smaller" than t_{xs}. This is evident in the sum example – shipping the partial sums z_i is more efficient than shipping the partial bags xs_i.

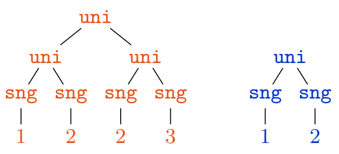


Fold Examples. The `fold` function provides a generic mold for specifying operations on collections. Aggregation functions like `min`, `max`, `sum`, and `count`, existential qualifiers like `exists` and `forall`, as well as collection processing operators like `map` and `filter` can be defined as folds.

Moreover, starting from `fold` we can define an algebraic structure known as *monad* on top of $\mathsf{Bag}\,A$ and enable declarative specification of computations.

2.3 Bag Comprehensions

Consider two bags $xs = \{\{1, 2, 2, 3\}\}$ and $ys = \{\{1, 2\}\}$ and their corresponding constructor application trees



in an example where you 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 as a set comprehension.

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

If xs and ys were lists, we could write a Python list comprehension.

```
[ (x, y) for x in xs for y in ys if x == y ]
```

If xs and ys were database relations, we could write a select-from-where query (effectively a SQL comprehension).

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

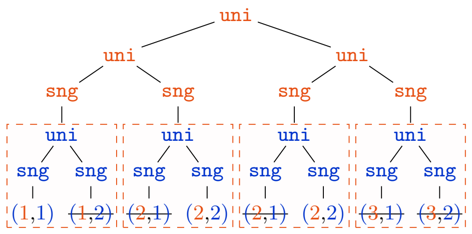
Finally, let xs and ys be values from the user-defined type $\mathsf{Bag}\,\mathbb{N}$ presented in Section 2.1. Modern functional languages like Scala allow us to use native comprehension syntax for arbitrary types, as long as those implement the so-called monad operators. We can therefore write the intended computation like this.

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

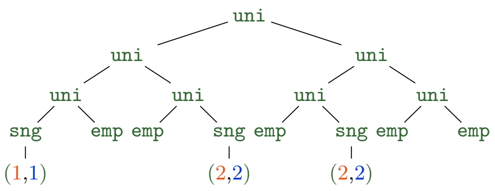
At parse time, the above comprehension is transformed into a chain of nested `flatMap` applications ending with a `map` and interleaved with `withFilter` applications as follows.

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

This desugaring scheme can be interpreted in terms of the structural recursion scheme discussed above. The `map` part of the `flatMap` application operates on the level of the (orange) xs tree – 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 (orange) 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 (see Section 3). Second, as a first-class citizen in an object language subject to metaprogramming, comprehensions can serve as an entry point for the integration of dataflow optimization techniques into general-purpose languages (see Section 4).

3. LANGUAGE DESIGN

Based on the formal foundations outlined in Section 2, we present Emma [2] – a DSL for parallel collection processing embedded in Scala. We first discuss the core API features by example, revisiting the issues outlined in Section 1, and then list the requirements of our approach to the host language.

3.1 Programming Abstractions

The core abstraction of our API is a generic type called `DataBag` which models bags in UNION-representation. The complete set of methods can be found in [4]. In the following, we discuss characteristic features by example.

For Comprehensions. Binary operators like `join` and `cross` are missing from the API. Instead, the `DataBag` type implements the monad operations discussed in Section 2.3. This allows us to write `Select-From-Where` expressions like the join from Section 1 in a declarative way.

```
for {
  email <- emails
  from  <- people
  to    <- people
  if from.email == email.from
  if to.email == email.to
} yield (from, to, email)
```

Folds. Computation on `DataBag` values is allowed only by means of structural recursion. To that end, we expose the `fold` operator from Section 2.2 as well as aliases for commonly used folds (e.g. `count`, `exists`, `minBy`). Counting the number of emails, for example, can be written as follows.

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

Nesting. The grouping operator introduces proper nesting:

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

The resulting bag contains groups of `values` that share the same `key`. Note that the `values` type is again `DataBag[A]`. This is fundamentally different from Spark, Flink, and Hadoop MapReduce, where the group values have the type `Iterable[A]` or `Iterator[A]`. An ubiquitous support for `DataBag` nesting allows us to hide the complexity of primitives like `groupByKey`, `reduceByKey`, and `aggregateByKey` behind a simple "`groupBy` and `fold`" programming model. For instance, calculating the `idf` term shown in Section 1 can be expressed as follows.

```
val idf = for {
  (t, docs) <- tf.groupBy{ case (_, t, _) => t }
} yield (t, math.log(N / docs.count()))
```

Due to the deep embedding approach we commit to, we can recognize nested `DataBag` patterns like the one above at compile time and rewrite them into more efficient equivalent expressions using primitives like `aggregateByKey`.

Coarse-Grained Parallelism Contracts. Current parallel dataflow APIs 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 its `DataBag` abstraction itself serves as a *coarse-grained* contract for data-parallel computation. The promise Emma gives 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, and (*iii*) take a holistic approach while translating them as parallel dataflows. This also allows to transparently insert primitives influencing execution like `broadcast` and `cache` as part of the compilation process.

3.2 Host Language Desiderata

The decision to base our implementation on Scala is motivated by purely pragmatic reasons: (*i*) Scala supports `for`-comprehensions for user-defined types, (*ii*) the runtimes we target have Scala APIs, and (*iii*) lightweight embedding and metaprogramming are enabled through Scala's macro and reflection facilities. In theory, however, any language which satisfies the above requirements can be used as a host-language for a similar compiler pipeline.

4. COMPILER PIPELINE

The basic compilation steps are depicted in Figure 1. At compile time, a Scala macro is used to (*i*) lift the Scala AST into a suitable intermediate representation (IR), (*ii*) apply logical rewrites that maximize data-parallelism to the Emma IR (see Section 4.1), and (*iii*) compile the result as a binary driver program with staged comprehensions. At runtime, Scala's reflection API is used to (*iv*) translate the

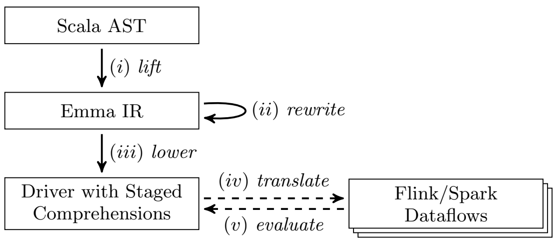


Figure 1: Basic Compiler Pipeline. Solid arrows represent static, and dashed dynamic (JIT) compilation.

staged comprehensions into the API of the targeted parallel dataflow engine, and to (*v*) evaluate those and feed the results back in the driver. In the rest of this section, we focus on some aspects of steps (*ii*) and (*iv*). The goal is to illustrate optimization techniques widely adopted by the database community on top of bag comprehensions as a model for program manipulation through metaprogramming. As in Section 3, more information can be found in [4].

4.1 Logical Optimizations

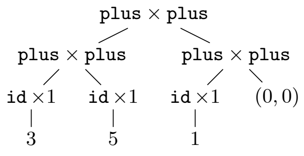
In this section, we show how two algebraic laws known as *banana-split* [6] and *fold-build fusion* [10] facilitate a logical rewrite upon generated groups. The rewrite transparently inserts partial aggregates whenever possible and thereby removes expensive group materializations.

Rewrite Candidates. Candidates for this rewrite are `groupBy` terms where (*i*) all occurrences of the group `values` are consumed by a `fold`, and (*ii*) these folds do not have data dependencies. When the optimization is triggered, the `groupBy` is replaced by an `aggBy` operator which fuses the group construction performed by the `groupBy` together with the subsequent `fold` applications on the group values. In terms of the APIs discussed in Section 1, this corresponds to replacing `groupBy` with `reduceByKey` whenever possible.

Banana Split. The banana split law generalizes the machinery behind loop fusion for arbitrary structural recursion. Informally, it states that a pair of folds can be rewritten as a fold over pairs. For example, the two folds below calculate a sum and a count over the same input collection.

```
val sum   = xs.fold(0, id, plus)
val count = xs.fold(0, x => 1, plus)
```

Those can be substituted by a single `fold` which operates by pairwise application of the original substitution functions. As a result, a pair containing the results of the two original folds can be computed in a single pass. To illustrate the idea, take a look at the application tree of the resulting pairwise `fold` over $\{\{3, 5, 1\}\}$.



Fold-Build Fusion. The second law enables a rewrite which in functional programming languages is commonly known as *cheap deforestation*. Intuitively, the law states that an operation that *constructs* a bag can be *fused together* (or in database terms – it can be *pipelined*) with a subsequent `fold` over the constructed value. To illustrate why `groupBy` can be seen as a build operator, consider the following naive definition of `tf.groupBy(...)`.

```
val tfGrp = for {
  t <- tf.map{ case (_, t, _) => t }.distinct()
} yield {
  val docs = tf.withFilter(_._2 == t)
  (t, docs)
}
```

We bind `t` over the set of distinct terms contained in `tf`, and pair each binding with its corresponding `docs` derived with a filter over `tf`. Substituting `tfGrp` in the code from Section 3.1, we consume the grouped result as follows.

```
val idf = for {
  (t, docs) <- tfGrp
} yield (t, math.log(N / docs.count()))
```

Knowing that the group values (`docs`) are used only in the context of a `count` application, we can build upon referential transparency and move the application up to `tfGrp`.

```
val tfAgg = for {
  t <- tf.map{ case (_, t, _) => t }.distinct()
} yield {
  val dfrq = tf.withFilter(_._2 == t).count()
  (key, dfrq)
}
```

The fold-build fusion law tells us that `withFilter` and the `count` can be fused to a single `fold` as follows.

```
val dfrq = tf.fold(0, if _._2 == t 1 else 0, plus)
```

The rewritten definition pairs terms directly with the aggregated document frequencies `dfrq` instead of materializing `docs`. To compensate for the rewrite at the consumer site, we remove the original `count` and directly refer to `dfrq`.

```
val idf = for {
  (term, dfrq) <- trAgg
} yield (term, math.log(N / dfrq))
```

In practice, we directly substitute `groupBy` with `aggBy`, but the comprehension model enables better understanding and reasoning about the soundness of such rewrites.

Other logical rewrites such as flattening [22] can also be performed at this step in order to maximize data-parallelism.

4.2 Dataflow Generation

As a result of the logical optimizations from Section 4.1, we obtain a modified AST for the original program. The JIT compiler then offloads the identified maximal `DataBag` terms to a parallel execution engine like Spark or Flink as promised. To illustrate the challenge here, consider again the (already normalized) join comprehension from Section 3.

```
for {
  email <- emails
  from  <- people
  to    <- people
  if from.email == email.from
  if to.email == email.to
} yield (from, to, email)
```

According to the semantics from Section 2.3 the above comprehension is equivalent to the following functional term.

```
emails.flatMap(email =>
  people.flatMap(from =>
    people
      .withFilter(to => from.email == email.from)
      .withFilter(to => to.email == email.to)
      .map(to => (from, to, email))))
```

A blunt translation approach would be to take the functional form and substitute the collection type from `DataBag` to either `RDD` or `DataSet`. This is not a good strategy for two reasons. First, it only allows for data-parallelism over the top-most collection (in this case `emails`), while everything else must be fully replicated across the cluster. Second, local evaluation amounts to brute-force nested loops.

Fortunately, the database community has solutions for this kind of problems. In relational databases, SQL queries are represented as expression trees called *logical plans*. For `Select-From-Where` queries, the leafs of the tree denote input relations, while inner nodes denote binary `join` applications. Due to the associativity and commutativity of `join`, however, multiple logical plans exist for the same query. Query optimizers use dynamic programming to pick an optimal plan based on a data-driven cost model.

To bridge the gap between the functional and the database world, we revise some old ideas from Grust [12] in the context of parallel dataflows. Observe that the abstract semantics of an (equi-)`join` operator can be defined in terms of a comprehension.

```
def join(k1, k2)(xs, ys) = for {
  x <- xs
  y <- ys
  if k1(x) == k2(y)
} yield (x, y)
```

Armed with that insight, we can devise recursive procedures that transform comprehension expressions into functionally closed cascades of joins. A bottom-up procedure may begin by joining `email` and `from`, and continue recursively until all generators are eliminated. This technique is known as "Selinger-style query optimization" [20].

```
// intermediate result (closed functional term)
val ir = join(_.from, _.email)(emails, people)
// final result (residual comprehension term)
for {
  (email, from)  <- ir
  to             <- people
  if to.email == email.to
} yield (from, to, email)
```

A top-down procedure, on the other side, may first split the original comprehensions in two, join their results, and continue recursively until all comprehensions have one generator. This technique is known as "Volcano-style query optimization" [11].

```
// intermediate result (residual comprehension term)
val ir = for {
  email <- emails
  from  <- people
  if from.email == email.from
} yield (from, email)
// final result (closed functional term)
join(_._2.to, _.email)(ir, people)
```

Either way, after the rewrite procedure terminates, we end up with an algebraic variant of the original comprehension based on second order combinators (like `join` and `flatMap`) in direct correspondence with the primitives offered by the targeted dataflow APIs. Translation from this form to the concrete target API can be done in a single traversal pass.

5. RELATED WORK

Object-Oriented Databases. A multitude of research has been performed during the OODBs era to bridge the gap between programming languages and database querying. Their primary goal was to make database access from the programming language transparent, focusing mainly on object persistence. OODBs attempted to support *all* the power of the host programming language (e.g., [16]) but did not succeed, mainly due to the complexity of such an undertaking. Another approach towards solving the impedance mismatch (i.e., using a string-quoted language like SQL within Java, C, Ruby, etc.) was PASCAL/R [14] – a first step towards integrated querying facility later resurrected by LINQ [18]. XQuery [7] (with scripting extensions for executing loops) provides a completely integrated programming and querying language that supports a flavor of comprehension syntax. However, XQuery is tied to the XML data model whereas our approach is oblivious to the data model.

More Recent Attempts. Similar to LINQ, Ferry [13] is a comprehensions-based programming language that facilitates database-supported execution of entire programs. To be evaluated, LINQ and Ferry programs both map into an intermediate form suitable for execution on SQL:1999-capable relational database systems. Similar to XQuery, Ferry, and LINQ, Emma is a comprehensions-based language, but targets JVM-based parallel dataflow engines. Moreover, the Emma compiler pipeline relies on a holistic view of the quoted code which simultaneously considers all comprehended terms.

Algebraic Approach. In spirit, the ideas presented here follow a line of work exploring the intersection between type theory, functional programming, and data management. The starting point is the work by Buneman et al. [8], who showed that monads can be used to generalize nested relational algebra to different types of collections and complex objects. The implications of using insert or union representation for parallel processing have been explored by Suciu and Wong [21] and more recently by Steele [15]. The approach of "comprehending dataflows" presented in Section 4.2 draws on ideas originally proposed by Grust [12]. Our aim is to highlight the importance of this line of work in facilitating seamless integration of declarative, data-parallel collection processing into a general-purpose host language.

To the best of our knowledge, Emma is the first DSL for parallel data analysis that promotes comprehensions as first-class citizen. The metaprogramming approach allows for hiding low-level API primitives behind a declarative, ubiquitous abstraction (`DataBag`) and maintaining competitive performance through a series of holistic optimizations.

6. SUMMARY & OUTLOOK

We illustrated common abstraction leaks shared between distributed collection processing APIs offered by state-of-the-art parallel dataflow engines. We argued that such leaks hinder the adoption of these APIs as a basic tool for advanced data analysis due to the burden imposed on the programmer. To alleviate this burden, we promoted the use of monad comprehensions over bags in union representation as first-class citizens in embedded DSLs. As a proof-of-concept, we presented Emma – a Scala DSL that offers (*i*) declarative dataflow syntax, and (*ii*) advanced rewrite-based optimizations. Emma programs can thereby employ dataflow engines such as Flink and Spark as transparent co-processors.

Embedding Approach. The ideas behind Emma require embedding in a general-purpose host language. While we currently rely on quotation-based embedding (as advocated by Lisp), type-based embedding (as advocated by LMS [19]) is an appealing alternative due to a more flexible metaprogramming infrastructure. Investigating and comparing the practical benefits of the two approaches poses an interesting research question.

Future Work. At the moment, the comprehensions discovered by the the Emma compiler are translated into target-engine dataflows in a heuristic manner. This approach is sub-optimal with respect to the optimization potential that can be harvested at runtime. To this end, we are currently working on an optimizer that will statically pre-compute interesting physical properties *across* dataflows and use this information in the JIT compilation phase. Finally, we are developing a linear algebra API that will allow for mixed use of bags, matrices, and vectors. Interested readers can learn more about Emma at our project webpage [2].

Acknowledgements. The authors would like to thank Sebastian Breß, Lukas Egger, Zachary Ives, and Tamara Mendt for reviewing a draft version of this article and providing insightful improvement suggestions.

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.

7. REFERENCES

[1] Cascading Project. `http://www.cascading.org/`.

[2] Emma Language. `http://www.emma-language.org/`.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[4] 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.

[5] K. S. Beyer, V. Ercegovac, R. Gemulla, A. Balmin, M. Y. Eltabakh, C. Kanne, F. Özcan, and E. J. Shekita. Jaql: A scripting language for large scale semistructured data analysis. *PVLDB*, 4(12):1272–1283, 2011.

[6] R. Bird and O. de Moor. *The Algebra of Programming*. Prentice Hall, 1997.

[7] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. Xquery 1.0: An XML query language, 2002.

[8] 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.

[9] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 2008.

[10] A. J. Gill and S. L. P. Jones. Cheap deforestation in practice: An optimizer for haskell. In *IFIP Congress (1)*, pages 581–586, 1994.

[11] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218. IEEE Computer Society, 1993.

[12] T. Grust. *Comprehending Queries (PhD Thesis)*. PhD thesis, Universität Konstanz, 1999.

[13] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. FERRY: database-supported program execution. In *SIGMOD Conference*, pages 1063–1066. ACM, 2009.

[14] M. Jarke and J. W. Schmidt. Query processing strategies in the PASCAL/R relational database management system. In *SIGMOD Conference*, pages 256–264. ACM Press, 1982.

[15] G. L. S. Jr. Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful. In *ICFP*, pages 1–2. ACM, 2009.

[16] C. Lamb, G. Landis, J. A. Orenstein, and D. Weinreb. The objectstore database system. *Commun. ACM*, 34(10):50–63, 1991.

[17] J. Lambek. Least fixpoints of endofunctors of cartesian closed categories. *Mathematical Structures in Computer Science*, 3(2):229–257, 1993.

[18] 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.

[19] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *GPCE*, pages 127–136. ACM, 2010.

[20] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD Conference*, pages 23–34. ACM, 1979.

[21] D. Suciu and L. Wong. On two forms of structural recursion. In *ICDT*, volume 893 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 1995.

[22] A. Ulrich and T. Grust. The flatter, the better: Query compilation based on the flattening transformation. In *SIGMOD Conference*, pages 1421–1426. ACM, 2015.

[23] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In E. M. Nahum and D. Xu, editors, *HotCloud*. USENIX, 2010.