Implicit Parallelism through Deep Language Embedding

Alexander Alexandrov Andreas Kunft Asterios Katsifodimos Felix Schüler
Lauritz Thamsen Odej Kao Tobias Herb Volker Markl

TU Berlin
firstname.lastname@tu-berlin.de

ABSTRACT

The appeal of MapReduce has spawned a family of systems that implement or extend it. In order to enable parallel collection processing with User-Defined Functions (UDFs), these systems expose extensions of the MapReduce programming model as library-based dataflow APIs that are tightly coupled to their underlying runtime engine. Expressing data analysis algorithms with complex data and control flow structure using such APIs reveals a number of limitations that impede programmer's productivity.

In this paper we show that the design of data analysis languages and APIs from a runtime engine point of view bloats the APIs with low-level primitives and affects programmer's productivity. Instead, we argue that an approach based on *deeply embedding* the APIs in a host language can address the shortcomings of current data analysis languages. To demonstrate this, we propose a language for complex data analysis embedded in Scala, which (*i*) allows for declarative specification of dataflows and (*ii*) hides the notion of data-parallelism and distributed runtime behind a suitable intermediate representation. We describe a compiler pipeline that facilitates efficient data-parallel processing without imposing runtime engine-bound syntactic or semantic restrictions on the structure of the input programs. We present a series of experiments with two state-of-the-art systems that demonstrate the optimization potential of our approach.

1. INTRODUCTION

One can argue that the success of Google's MapReduce programming model [7] is largely due to its expressiveness and simplicity. Exposing an API built around second-order functions like `map` f and `reduce` h enables general-purpose programming with collections via user-defined functions 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 program scalability.

Vanilla MapReduce is a perfect fit for generalized processing and aggregation of a single collection of complex objects, but programmers hit a barrier when trying to express more complex programs. A well-known example is coding a join in a MapReduce framework like Hadoop, which would require either encoding lineage in the `map` output or using auxiliary primitives like a distributed cache. Committing to either of these strategies would mean hard-coding a join parallelization strategy such as a repartition or a broadcast in the user code. This may cause performance degradations when the relative size of the two inputs changes. Moreover, algorithms for deep data analysis are characterized by non-trivial data and control flow structure. Again, a naïve approach where this structure is realized as a separate "driver" program neglects substantial optimization potential.

To overcome these limitations without sacrificing the benefits of seamless integration of UDFs and driver logic in a general-purpose host language like Java or Scala, projects like Cascading [1], HaLoop [8], SCOPE [14], Spark [35], and Stratosphere/Flink [6] have proposed various extensions of the MapReduce model. The bulk of these extensions facilitate one of following:

- **Rich dataflow APIs**. This includes providing more second-order constructs (e.g. `cogroup`, `cross`, `join`) as well as means to arrange them in an unrestricted way to allow advanced dataflows [1, 6, 14, 35].
- **Support for non-trivial data and control flow**. This includes primitives for efficient data exchange between the driver and the UDFs, as well as primitives for control flow (e.g. iterations) [8, 18, 35].

In each of these languages, these primitives directly correspond to runtime features, which leads to significant performance improvement. This evolution, however, also has its downsides. A critical look from a programmer's perspective reveals a new set of problems. To highlight them we use two code snippets showing the main loop of the k-means clustering algorithm implemented in Spark (Listing 1) and Flink (Listing 2).

Broadcast Variables. First, to find the nearest centroid for each point, both implementations use a `map` over the set of points (line 4): the `findNearestCntrd` UDF matches a point `p` against each centroid `c` and emits the `(c, p)` pair with minimal distance between them. To make the current set of centroids available to all UDF instances, both implementations use a special "broadcast" primitive which minimizes the shipping cost for read-only variables (line 10 for Spark, line 4 for Flink). The decision when to use this primitive is left to the programmer. Certainly, thinking

Listing 1: Spark k-means (simplified)

```
1   ... // initialize
2   while (theta) {
3     newCntrds = points
4       .map(findNearestCntrd)
5       .map( (c, p) => (c, (p, 1L)) )
6       .reduceByKey( (x, y) =>
7         (x._1 + y._1, x._2 + y._2) )
8       .map( x => Centroid(x._1, x._2._1 / x._2._2) )
9
10    bcCntrs = sc.broadcast(newCntrds.collect())
11  }
```

Listing 2: Flink k-means (simplified)

```
1   ... // initialize
2   val cntrds = centroids.iterate(theta) { currCntrds =>
3     val newCntrds = points
4       .map(findNearestCntrd).withBcSet(currCntrds, "cntrds")
5       .map( (c, p) => (c, p, 1L) )
6       .groupBy(0).reduce( (x, y) =>
7         (x._1, x._2 + y._2, x._3 + y._3) )
8       .map( x => Centroid(x._1, x._2 / x._3) )
9
10    newCntrds
11  }
```

about such runtime aspects is a source of distraction that affects productivity, especially when working on complex algorithms. It is therefore desirable that they be hidden from the developer and handled transparently in the background.

Partial Aggregates. Second, the code for computing the new centroids is distributed across three different UDFs: extending the `(c, p)` tuple with an extra `count` = 1 (line 5), grouping the result triples by `c` and summing up the `sum` and `count` values (line 6), and, finally, computing the new mean as the ratio of the two aggregates (line 8). The reason for this verbosity is the same across all data-parallel systems: computations that can be pushed behind a grouping operation have to be identified as such explicitly through special language primitives (e.g., `combine` in MapReduce, `reduceByKey` in Spark, and `reduce` in Flink). For example, putting the centroid computation in a single UDF:

$$h : (m,\ pts) \mapsto (m,\ \texttt{sum}(pts)/\texttt{count}(pts)) \qquad (\textsc{Ctrds})$$

would result in execution plans that materialize the groups before they are aggregated in a `map` h application. This would impact both resource usage and performance, as shuffling and merging partial aggregates requires less memory and bandwidth than shuffling and merging the corresponding partial groups. Indeed, one of the most important points stressed in the programming guides of all languages is the need to recognize and phrase such aggregations using the corresponding construct. Again, we argue that optimizations tied to the execution model should be performed transparently in the background.

Native Iterations. The third problem we highlight is concerned with the form of loop primitives (line 2). While Spark uses the native `while` loop of the host language (Scala), Flink requires a dedicated `iterate` construct. The reason for this difference is once more tied to the execution modes employed by the two systems. Spark only supports acyclic dataflows and realizes loops by lazily unrolling and evaluating dataflows inside the loop body. Flink, in contrast, has native runtime support for iterations. This approach has performance benefits, but requires special feedback edges in the dataflow plans. Since these plans are constructed only via API methods, the only way to expose native iterations is with a dedicated higher-order construct like `iterate`. Ideally, however, runtime and language aspects should be separated and programmers should be able to use a `while` loop in both cases.

The aforementioned observations suggest a runtime-centric evolution of these languages. Tied to the original decision to *embed them shallowly* in the host language as pure libraries, this has, over time, caused a significant increase in their complexity. This is apparent by the amount of primitives tied to runtime features that can be found across all languages at the moment.

In this paper, we argue that an approach based on *deep embedding* will solve this problem and that taking a holistic view over the abstract syntax tree of data analysis programs at compile time would allow us to introduce several degrees of freedom that will result in non-trivial program optimization.

The contributions of this paper are summarized as follows:

- We propose Emma – a language for parallel data analysis that is deeply embedded in Scala. Emma compiles data-analysis programs holistically using an intermediate representation (IR) based on [22]. This facilitates:

  – *Parallelism Transparency.* Dataflows can be specified using `for`-comprehensions – a declarative syntax akin to SQL which also natively supports nesting. Language artifacts that suggest data-parallelism like the algebraic (functional) style of dataflow assembly seen today or the explicit movement of data between the local and distributed runtime can be hidden from the programmer.

  – *Advanced Optimizations.* Comprehending nested dataflow expressions enables non-trivial algebraic optimizations, like rewriting the centroids computation from the form shown in Ctrds into the form seen in Listings 1 and 2. In addition, a holistic view of a data-analysis program as a mix of parallel datflows and centralized control-flow, allows for a range of physical optimizations.

- We present a compiler pipeline implemented in Scala that can generate code for multiple runtime engines.

- We provide a set of experiments showing that our compiler pipeline and optimizations enable declarative specification of programs while matching the performance of hand-coded programs on low level APIs.

The remainder of this paper is structured as follows. Section 2 gives some necessary technical background. Section 3 presents the proposed language, while Section 4 sketches the associated compiler pipeline and explains the steps we take to realize the degrees of freedom listed above. Section 5 presents the results of an experimental evaluation of the presented ideas. Section 6 reviews related work, and, Section 7 summarizes and discusses ideas for future work.

2. PRELIMINARIES

This section first presents a set of desired properties for data analysis languages (Section 2.1) and then reviews the main concepts that lay the foundation for our work (Section 2.2).

2.1 Desiderata

Our goal is to embed parallel dataflows in a host language transparently and without sacrificing performance. Complex rewritings require access to the full program at compile time. To this end, we need an intermediate representation that simplifies the rewriting process and provides means for reasoning about optimization. Based on the problems observed in the k-means excerpts for Spark and Flink, we formulate the following list of desiderata for a language that aims to facilitate advanced data analysis at scale.

D1 Declarative Dataflows. Formulating dataflows with higher-order functions can be tedious because the primitives (e.g. `cogroup`, `combine`, `reduce`) define the units of parallelism *explicitly.* This impedes productivity by forcing the programmer into constantly thinking about data-parallelism. We believe that a more declarative program specification should be preferred, and that the language should ensure optimal parallel execution.

D2 Transparent Execution Engine. The notion of a (secondary) parallel runtime running next to the (primary) host language runtime should be hidden from the programmer. The decision to offload certain expressions to the parallel runtime should be transparent.

To achieve D1, we need a declarative programming abstraction for parallel dataflows. One possible strategy is to use an SQL dialect and quote SQL-like expressions (similar to JDBC statements). Another strategy is to design an API with fluent syntax on collections (similar to LINQ [27] or Cascading [1]). In our language design, we keep the collection type, but hide the manipulation primitives (e.g., `cogroup`, `combine`, `reduce`) behind high-level language features with which users are familiar. To this end, we make use of the fact that Scala, our host language of choice, natively supports a variant of comprehension syntax – a declarative construct that can be used for dataflow specification. Comprehension syntax is found, under different syntactic sugar, in a growing number of languages (e.g. C# 3.0, Erlang, Ruby, Haskell, Python, Perl 6).

To achieve D2, we have to detect and inject the necessary calls to the parallel execution engine at compile time. Scala macros [11] – a recent addition to Scala that allows meta-programming at the level of abstract syntax trees – provide tools to realize a compiler pipeline that implements D2 without modifying the host language compiler. The details of the proposed compiler pipeline are presented in Section 4.

2.2 Intermediate Language Representation

Our ultimate goal is to hide the notion of parallelism behind simple, declarative abstractions. To achieve this without sacrificing performance, we need to detect and optimize dataflow expressions at compile time.

A straight-forward approach is to base the compiler logic entirely on top of the IR provided by our host language – Scala abstract syntax trees (ASTs). Reasoning about the soundness and performing dataflow transformations directly on top of Scala ASTs, however, can be tedious to follow and cumbersome to implement.

Instead of following this path, we propose a layered intermediate representation where dataflow expressions found in the original AST are converted and transformed into a declarative, calculus-like representation called *monad comprehensions.* Monad comprehensions combine concepts from functional programming and database systems under the same theoretical framework and have already been exploited as IR of database queries in the past [9, 20, 22].

For the discussion of monad comprehensions in the end of this section (Section 2.2.3), we need to first introduce abstract data types (ADTs) as a model for the structure and the semantics of bags (Section 2.2.1), and structural recursion as a model for computations on bags (Section 2.2.2).

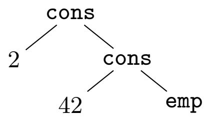
2.2.1 Bags as Algebraic Data Types

The core abstractions provided by Flink – DataSet, Spark – RDD[1], Cascading – Collection and LINQ – IEnumerable [27] represent a homogeneous collection with bag semantics – i.e., elements that share the same type, do not have a particular order, and duplicates are allowed. To motivate our choice of intermediate representation, we first study the structure and semantics of bags.

Bag Structure. We can use the theory of *algebraic specifications* to formally model the structure and semantics of the type of bags [10, 17]. To specify bags algebraically (i.e., using functions), we first define a *constructor algebra*:

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

The above definition states that the values of the polymorphic type $\mathsf{Bag\,A}$ – i.e., all bags with elements from A – are *identified* with corresponding *constructor application trees.* As the name suggests, these trees denote applications of the constructor functions `emp` (which denotes the empty bag), and `cons` (which denotes the bag where the element x is added to the bag xs). For example, the bag of integers $\{\!\{\, 2, 42 \,\}\!\}$ is identified with the constructor application tree:



Bag Semantics. By definition, constructor algebras like AlgBag-Ins are *initial* and thereby, following Lambek's lemma [26], *bijective.* What does this mean? Essentially, it means that the association between trees and values is bidirectional – each constructor application tree t_{xs} represents precisely one bag value xs and vice versa. This poses a problem, as it contradicts our intended semantics, which state that the element order should be arbitrary. 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*:

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

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 tree 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 $[\, \{\!\{\, 2, 42 \,\}\!\} \,]$.

[1] An acronym for Resilient Distributed Datasets.

Relevance for Data Management. An algebraic view on bags is relevant from a database systems perspective. Conceptually, the $xs \mapsto t_{xs}$ direction can be interpreted as a recursive parser that decomposes a bag xs into its constituting elements x_i [20]. Essentially, the same logic is implemented by the database `scan` operator. Indeed, we can realize a simple iterator-based version of `scan` with the help of the AlgBag-Ins constructors (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 t_{xs} 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*:

$$\begin{array}{lll} \mathsf{type\ Bag\ A} & = \texttt{emp} & \\ & \mid\ \texttt{sng}\, x\colon \mathsf{A} & (\textsc{AlgBag-Union}) \\ & \mid\ \texttt{uni}\, xs\colon \mathsf{Bag\ A}\ ys\colon \mathsf{Bag\ A} & \end{array}$$

$$\begin{array}{rcll} \texttt{uni}\, xs\, \texttt{emp} & = & \texttt{uni}\, \texttt{emp}\, xs\ =\ xs & (\textsc{EQ-Unit}) \\ \texttt{uni}\, xs\ (\texttt{uni}\, ys\, zs) & = & \texttt{uni}\ (\texttt{uni}\, xs\, ys)\, zs & (\textsc{EQ-Assoc}) \\ \texttt{uni}\, xs\, ys & = & \texttt{uni}\, ys\, xs & (\textsc{EQ-Comm}) \end{array}$$

The `emp` operator creates the empty bag $\{\!\{\}\!\}$, `sng` x creates a singleton bag $\{\!\{x\}\!\}$, and `uni` $xs\ ys$ creates the union bag of xs and ys.

A translation from AlgBag-Ins to AlgBag-Union and vice versa follows immediately from the initialilty property. Bags modeled in AlgBag-Union representation, however, are a more natural fit in scenarios where the bag contents are distributed accross multiple nodes, as we will see in Section 2.2.2. We therefore rely on AlgBag-Union for the language presented in Sections 3 and 4.

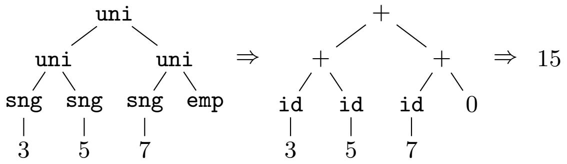
2.2.2 Structural Recursion on Bags

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 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 of a bag $xs = \{\!\{3, 5, 7\}\!\}$. We can define this operation with a higher-order function called `fold` which implements structural recursion on AlgBag-Union-style trees:

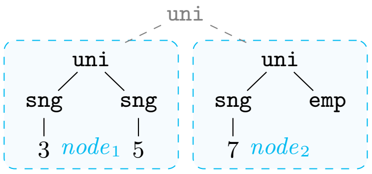
```
// fold: structural recursion on union-style bags
def fold[A,B](e: B, s: A=>B, u: (A,A)=>B)
             (xs: Bag[A]) = xs match {
  case emp => e
  case sng(x)  => s(x)
  case uni(ys,zs) => u(fold(e,s,u)(ys),fold(e,s,u)(zs))
}
```

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 use $\texttt{e} = 0$, $\texttt{s} = \texttt{id}$, and $\texttt{u} = +$:

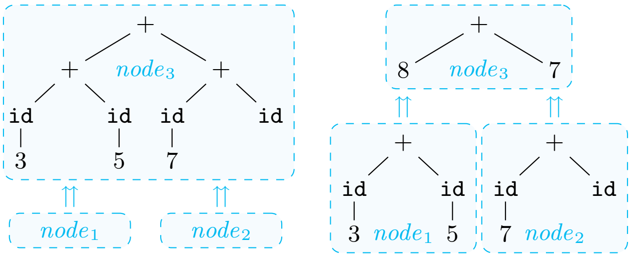


In this case, each `u`-node represents the partial summation of all integers occurring at the leafs below it.

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 $xs = \texttt{uni}\, xs_1\, xs_2$, 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 `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 that some loss of information occurs when evaluating the substituted t_{xs} tree 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 much more efficient than shipping the partial bags xs_i:



Well-Definedness Conditions. Under which conditions is a `fold` well-defined? Since t_{xs} is an arbitrary tree from $[t_{xs}]$, we want to ensure that the `fold` result is the same for all equivalence class members. To achieve this, we impose on `e`, `s`, and `u` the same equations as we did on `emp`, `sng`, and `uni` in AlgBag-Union:

$$\begin{aligned} \texttt{u}(x,\, \texttt{e}) = \texttt{u}(\texttt{e},\, x) &= x \\ \texttt{u}(x,\, \texttt{u}(y,\, z)) &= \texttt{u}(\texttt{u}(x,\, y),\, z) \\ \texttt{u}(x,\, y) &= \texttt{u}(y,\, x) \end{aligned}$$

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`, `flatMap`, `filter`, and `join` can be defined using only `fold` instances.

More interestingly, `map`, `flatMap`, and `filter` together with the Bag algebra itself constitute an algebraic structure

Listing 3: DataBag API

```
class DataBag[+A] {
  // Type Conversion
  def this(s: Seq[A]) // Scala Seq -> DataBag
  def fetch() // DataBag -> Scala Seq
  // Input/Output (static)
  def read[A](url: String, format: ...): DataBag[A]
  def write[A](url: String, format: ...)(in: DataBag[A])
  // Monad Operators (enable comprehension syntax)
  def map[B](f: A => B): DataBag[B]
  def flatMap[B](f: A => DataBag[B]): DataBag[B]
  def withFilter(p: A => Boolean): DataBag[A]
  // Nesting
  def groupBy[K](k: (A) => K): DataBag[Grp[K,DataBag[A]]]
  // Difference, Union, Duplicate Removal
  def minus[B >: A](subtrahend: DataBag[B]): DataBag[B]
  def plus[B >: A](addend: DataBag[B]): DataBag[B]
  def distinct(): DataBag[A]
  // Structural Recursion
  def fold[B](z: B, s: A => B, p: (B, B) => B): B
  // Aggregates (aliases for various folds)
  def minBy, min, sum, product, empty, exists, ...
}
class StatefulBag[A <: Key[K], K] {
  // Stateful Conversion
  def this(s: DataBag[A]) // DataBag -> StatefulBag
  def bag(): DataBag[A] // StatefulBag -> DataBag
  // Point-wise update (with and w/o update messages)
  def update(u: A => Option[A]): DataBag[A]
  def update[B <: Key[K]](messages: DataBag[B])
    (u: (A, B) => Option[A]): DataBag[A]
}
```

known as *monad*. The next section describes a syntactic abstraction over monads that is ideally suited for the needs of our intermediate language.

2.2.3 Monad Comprehensions

In order to describe the set of all pairs that satisfy a certain predicate p, one would most probably use set comprehension syntax:

$$\{\ (x, y) \mid p(x, y),\ x \in X,\ y \in Y\ \}$$

Although monad comprehensions are not widely used by data processing systems today, we believe that their connection with data-parallel computations like `fold` and `map` makes them a natural fit for large-scale processing.

Anatomy. Following Grust's notation and nomenclature [20], a *monad comprehension* has the general form:

$$[\![\ e \mid qs\ ]\!]^{\mathsf{T}} \qquad \text{(MC)}$$

Here, T is the monad type, e is called the *head* of the comprehension, and qs is a sequence of *qualifiers*. Qualifiers can be either *generators* of form $x \leftarrow xs$, or *filters* of form $\mathtt{p}\ x_i$. For instance, a join expression $xs \bowtie_{\mathtt{p}} ys$ can be written in comprehension syntax as:

$$[\![\ (x, y) \mid x \leftarrow xs,\ y \leftarrow ys,\ \mathtt{p}\,x\,y\ ]\!]^{\mathsf{Bag}} \qquad \text{(MC-Ex1)}$$

The above expression should be "comprehended" intuitively as follows: for each x from xs and y from ys where $\mathtt{p}\,x\,y$ is true, emit (x, y) tuples and use them to construct an AlgBag-Union tree that represents the result bag.

Comprehending Folds. The comprehension language discussed so far allows us to formally capture the `Select-From-Where` fragment of the relational algebra using a declarative syntax similar to SQL and the relational calculus. Grust also suggested a way to represent folds as comprehensions over identity monads with zero. Continuing the example from above, we can represent the aggregation $\gamma_{\mathtt{SUM}\,y}(xs \bowtie_{\mathtt{p}} ys)$ as:

$$[\![\ y \mid x \leftarrow xs,\ y \leftarrow ys,\ \mathtt{p}\,x\,y\ ]\!]^{\mathtt{fold(0,id,+)}} \qquad \text{(MC-Ex2)}$$

The above expression should be "comprehended" intuitively as follows: for each x from xs and y from ys where $\mathtt{p}\,x\,y$ is true, emit the y's and use them to construct and evaluate a $(0, \mathtt{id}, +)$-tree that represents the resulting sum.

Relevance for Data-Parallel Languages. From a programmer's perspective, the main advantage of monad comprehensions is their declarative syntax. We make use of this feature in the design of our language (Section 3.1). From a compiler's perspective, the main advantage of monad comprehensions is the fact that they allow nesting. For example, the head expression e of a comprehension c_1 may contain another comprehension c_2 which refers to variables bound in c_1. This allows us to formulate and execute non-trivial unnesting rewrites (Section 4.2).

3. LANGUAGE DESIGN

As illustrated in Section 1, the current set of languages for data-parallel analysis with UDFs have a number of issues. In this section, we present Emma – a novel language designed against the desiderata outlined in Section 2.1 that resolves these issues.

3.1 Programming Abstractions

The main abstraction of our language is similar to the one used by Spark and Flink – a type that represents homogeneous collections with bag semantics called `DataBag`.The `DataBag` abstraction API can be found in Listing 3. In the following, we discuss the supported `DataBag` operators.

Declarative SPJ Expressions. The reader may notice that some basic binary operators like `join` and `cross` are missing from the API. This is a language design choice; instead, we just provide the monad operations `map`, `flatMap`, and `withFilter` mentioned in Section 2.2.2. Through that, we implicitly enable a flavor of monad comprehensions known as `for` comprehensions in Scala's concrete syntax (see §6.19 in [2]). `Select-Project-Join` expressions like the one in MC-Ex1 can be then written in a declarative way:

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

Folds. Native computation on `DataBag` values is allowed only by means of structural recursion. To that end, we expose the `fold` operator from Section 2.2.2 as well as aliases for commonly used folds (e.g. `count`, `exists`, `minBy`). Summing up a bag of numbers, for example, can be written as:

```
val z = xs.fold(0, x => x, (x, y) => x + y)
val z = xs.sum() // alias for the above
```

Nesting. The grouping operator introduces nesting:

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

The resulting bag contains groups of input elements that share the same key. The `Grp` type has two components – a group `key`: `K`, and group `values`: `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 primitives like `groupByKey`, `reduceByKey`, and `aggregateByKey`, which might appear to be the same, behind a uniform "`groupBy` and `fold`" programming model. To group a bag of (a, b) tuples by a and compute the count for each group, for example, we can write:

```
for (g <- xs.groupBy(_.a)) yield g.values.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` (for more details, see Section 4.2.2).

Interfacing with DataBags. To interface with the core `DataBag` abstraction, the language also provides operators to read and write data from a file system (line 5, Listing 3), as well as convertors for Scala `Seq` types (line 2).

Stateful Bags. A range of algorithms require iterative bag refinement via point-wise updates. In domains where these algorithms occur often, this manifests as domain-specific programming models like "vertex-centric" for graph processing. In Emma, we capture these cases in a domain-agnostic way through a core abstraction for "stateful" bags. Conversion between (stateless) `DataBag` and `StatefulBag` instances is enforced explicitly by the user (lines 25-26). The elements of the `StatefulBag` can be updated in-place with a UDF which either operates on an element alone (line 28) or takes an associated update message that shares the element key (line 29). The UDF decides whether to change an element and optionally returns its new version as an `Option[A]` instance. Modified elements are transparently merged in the current bag state and forwarded to the client as a `DataBag[A]` representing the changed delta. This allows native integration of both naive and semi-naive iterative dataflows in the core language (see the graph algorithms in Appendix A).

Coarse-Grained Parallelism Contracts. Data-parallel dataflow APIs typically 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. Emma gives the promise that it will (*i*) discover all maximal `DataBag` expressions in a code fragment, (*ii*) rewrite them logically in order to maximize the available degrees for data-parallelism, and (*iii*) take a holistic approach while translating them as concrete data-parallel dataflows.

Host Language Execution. The operators presented in Listing 3 are not abstract. The semantics of each operator are given directly as method definitions in Scala. This feature facilitates rapid prototyping, as it allows the programmer to incrementally develop, test, and debug the code at small scale locally as a pure Scala program.

3.2 Example: K-Means Revisited

To get a feeling for the language, we show a complete example of Lloyd's algorithm for k-means clustering in Listing 4. The core algorithm logic is implemented in lines 8-42. As the title of the paper promises, nothing in this range of lines suggests that the code will run in parallel. The programmer can focus on the core algorithm logic and use the native Scala implementation of our `DataBag` abstraction to debug and refine the code incrementally.

When the algorithm is ready, the code needs to be wrapped in special `parallelize` brackets (line 6). The bracketed code is rewritten by a Scala macro at compile time. The macro identifies maximal `DataBag` expressions (marked with blue dashed rectangles), rewrites them, and transforms them to parallel dataflow assembly code. The resulting program is wrapped in an `Algorithm` object that can be executed on different runtime engines (line 45). In the next section, we discuss the compiler pipeline implemented by the `parallelize` macro in detail.

Listing 4: k-means in Emma

```
1  // define schema
2  case class Point(id: Int, pos: Vector[Double])
3  case class Solution(cid: Int, p: Point)
4
5  // define algorithm
6  val algorithm = parallelize {
7    // read initial points
8    val points = read(inputUrl, CsvInputFormat[Point])
9    // initialize centroids
10   var ctrds = DataBag(for (i <- 1 to k) yield /*...*/)
11
12   // iterate until convergence
13   var change = 0.0; var epsilon = /* ... */;
14   while (change > epsilon) {
15     // compute new clusters
16     var clusters = (for (p <- points) yield {
17       val c = ctrds.minBy(distanceTo(p)).get
18       Solution(c.id, s.p)
19     }).groupBy(_.cid)
20     // compute new centroids
21     val newCtrds = for (clr <- clusters) yield {
22       val sum = clr.values.map(_.p.pos).sum()
23       val cnt = clr.values.map(_.p.pos).cnt()
24       Point(c.key, sum / cnt)
25     }
26     // compute the total change in all centroids
27     change = {
28       val distances = for (
29         x <- ctrds;
30         y <- newCtrds; if x.id == y.id) yield dist(x, y)
31       distances.sum()
32     }
33     // use new centroids for the next iteration
34     ctrds = newCtrds
35   }
36
37   // compute and write final solution
38   write(outputUrl, CsvOutputFormat[Solution])(
39     for (p <- points) yield {
40       val c = ctrds.minBy(distanceTo(p)).get
41       Solution(c.id, s.p)
42   })
43 }
44 // run on a parallel runtime (e.g. Spark, Flink)
45 algorithm.run(runtime.EngineX(host, port))
```

3.3 Host Language Choice

The decision to base our implementation on Scala is motivated by purely pragmatic reasons: (*i*) Scala supports `for`-comprehensions in its concrete syntax; (*ii*) the runtimes we target have Scala APIs, and (*iii*) complex compile-time rewrites are feasible through Scala macros. The point we want to stress in this paper, however, are the benefits of comprehensions over bags in UNION-representation as core abstraction for data-parallel programming: exposing this abstraction in the concrete language syntax facilitates declarativity and nesting. At the same time, promoting comprehensions to first-class citizens at compile-time allows a range of non-trivial optimizations. These optimizations are either not attainable or cumbersome to implement by state-of-the-

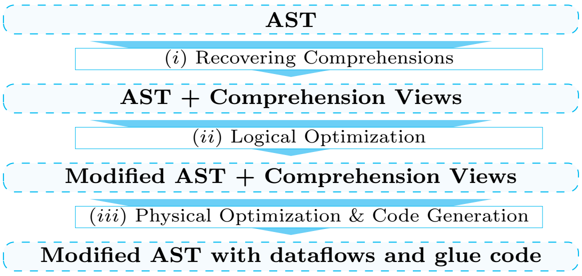

AST
(i) Recovering Comprehensions
AST + Comprehension Views
(ii) Logical Optimization
Modified AST + Comprehension Views
(iii) Physical Optimization & Code Generation
Modified AST with dataflows and glue code

Figure 1: `parallelize` pipeline

art parallel dataflow APIs. 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 steps taken by the `parallelize` macro are depicted in Figure 1. At step (*i*), the compiler processes the user code AST and constructs comprehension views over all data-parallel expressions (i.e., operations on `DataBag` instances). At step (*ii*), the AST and the associated comprehension views are rewritten logically in order to remove redundant group materializations and maximize the potential for data-parallelism. At step (*iii*), the rewritten comprehensions are transformed as algebraic terms over higher-order functions, effectively an abstract version of the syntax accepted by the underlying target runtimes. The original expressions are then substituted with calls to a just-in-time dataflow compiler. At runtime, the compiler assembles and executes the actual dataflows out of the derived higher-order terms. Holistic decisions about physical execution aspects can be thereby deferred to a point where the execution context is available. Although our approach is currently based on heuristics, the strategy allows us to introduce cost-based decisions in the future. In the following sections, we describe the most important aspects of each step.

4.1 Recovering Comprehensions

In the first step of the proposed compiler pipeline we construct an auxiliary view over all maximal `DataBag` expressions in the input AST.

Desugaring Comprehensions. As already stated, we use monad comprehensions as core abstraction both in the programming language (to facilitate declarativity) and in the auxiliary view during compilation (to facilitate program rewrite). Unfortunately, Scala's `for` comprehensions are specified as syntactic sugar which "desugars" as a chain of the underlying monad operations (`map`, `flatMap`, and `withFilter`) at AST construction time. For example, the `for` comprehension from line 28 is desugared into the equivalent term:

```
val distances = ctrds
  .flatMap(x => newCtrds
    .withFilter(y => x.id == y.id)
    .map(y => dist(x, y)))
```

Moreover, the programmer can also hard-code calls of `map`, `flatMap`, or `withFilter` in the input (e.g., the `map` calls used for projection in lines 22 and 23). To handle these situations, we propose a two step approach that first identifies maximal terms that can be comprehended, and then constructs and normalizes the associated comprehension for each such term.

Finding Comprehendable Terms. The algorithm for finding maximal comprehendable terms works as follows: we first collect all function application nodes that belong to the `DataBag` API, and then, based on the possible occurrence patterns defined by the `for` comprehension desugaring scheme, we merge all terms that will be comprehended with their parent. The statement in line 21 of Listing 4, for instance, will produce three terms: one for the outer `for` comprehension that binds the `clr` variable, and two for the aggregates on lines 22 and 23. The algorithm returns a set of nodes (corresponding to subtrees in the AST), which we see as comprehensions in order to optimize. Constructing a comprehension for these subtrees is broken into three substeps, namely: inlining, resugaring, and normalization.

Inlining. As a preprocessing step, we also inline all value definitions whose right-hand side is comprehended and referenced only once. This results in bigger comprehensions and increases the chances of discovering and applying comprehension level rewrites in the subsequent steps.

Resugaring Comprehensions. In the first step, we recursively traverse each term and "re-sugar" comprehensions using the following translation scheme (MC^{-1}):

$$\begin{aligned}
t'.\texttt{map}(x \mapsto t) &\Rightarrow \llbracket\, t \mid x \leftarrow \mathrm{MC}^{-1}(t')\,\rrbracket^{\mathsf{Bag}}\\
t'.\texttt{withFilter}(x \mapsto t) &\Rightarrow \llbracket\, x \mid x \leftarrow \mathrm{MC}^{-1}(t'),\, t\,\rrbracket^{\mathsf{Bag}}\\
t'.\texttt{flatMap}(x \mapsto t) &\Rightarrow \texttt{flatten}\,\llbracket\, t \mid x \leftarrow \mathrm{MC}^{-1}(t')\,\rrbracket^{\mathsf{Bag}}\\
t'.\texttt{fold}(e, s, u) &\Rightarrow \llbracket\, x \mid x \leftarrow \mathrm{MC}^{-1}(t')\,\rrbracket^{\texttt{fold}(e,s,u)}
\end{aligned}$$

Normalization. In the second step, we normalize the resulting comprehensions using the following set of rewrite rules:

$$\begin{aligned}
\texttt{flatten}\,\llbracket\,\llbracket\, e \mid qs'\,\rrbracket \mid qs\,\rrbracket^{\mathsf{T}} &\Rightarrow \llbracket\, e \mid qs,\ qs'\,\rrbracket^{\mathsf{T}}\\
\llbracket t \mid qs,\ x \leftarrow \llbracket t' \mid qs' \rrbracket,\ qs''\,\rrbracket^{\mathsf{T}} &\Rightarrow \llbracket t[t'/x] \mid qs,\ qs',\ qs''[t'/x]\rrbracket^{\mathsf{T}}\\
\llbracket\, e \mid qs,\ \llbracket\, p \mid qs''\,\rrbracket^{\texttt{exists}},\ qs'\,\rrbracket^{\mathsf{T}} &\Rightarrow \llbracket\, e \mid qs,\ qs'',\ p,\ qs'\,\rrbracket^{\mathsf{T}}
\end{aligned}$$

The first of these rules unnests a nested (inner) comprehension occurring in a (outer) comprehension head, while the second unnests a comprehension occurring in a generator. It is interesting to note that the second rule performs an optimization known as "fusion" at compile time. This optimization ensures that `map` and `fold` chains are compacted and executed in a single task and is otherwise achieved (at the expense of virtual function calls) via pipelining and operator chaining by the runtime. Finally, the third rule unnests `exists`-style nested comprehensions as shown by Grust in [22], and can be seen as a generalization of Kim's *type N* optimization [25].

Example. The `distances` expression from line 28 will be inlined in `distances.sum()` at line 31, and the functional term shown earlier will be "re-sugared" with the MC^{-1} scheme to create an associated comprehension:

$$\begin{aligned}
&\texttt{flatten}\,\llbracket\\
&\quad \llbracket\, \mathit{dist}(x, y) \mid\\
&\qquad y \leftarrow \llbracket\, y \mid y \leftarrow \texttt{ctrds},\ x.\mathit{id} = y.\mathit{id}\,\rrbracket\,\rrbracket \mid\\
&\quad x \leftarrow \texttt{newCtrds}\,\rrbracket^{\mathsf{sum}}
\end{aligned}$$

After normalization, the comprehension will look like:

$$\llbracket\, \mathit{dist}(x, y) \mid x \leftarrow \texttt{newCtrds},\ y \leftarrow \texttt{ctrds},\ x.\mathit{id} = y.\mathit{id}\,\rrbracket^{\mathsf{sum}}$$

The comprehended parts of the k-means code are highlighted with dashed rectangles on Listing 4.

Comprehension Views. The final result of these series of operations is a secondary layer imposed on top of the

AST which serves as a prism that provides "comprehension views" over all DataBag expressions. This view is the driving intermediate representation for the rest of the compiler pipeline.

4.2 Logical Optimizations

The derived comprehensions can be used as a basis for various logical optimizations that maximize the degree of data-parallelism exposed to the dataflow runtime. In this section, we first review the impacts of the unnesting step described in Section 4.1. Upon that, we show how an algebraic law known as *banana-split* facilitates the application of *fold-build-fusion* upon generated groups irrespective of the number and placement of group consuming folds. This provides a sound rewrite theory which enables *transparent insertion* of partial aggregates whenever possible.

4.2.1 Unnesting

Imagine a situation where a collection of emails has to be cross-checked against a list of blacklisted servers. One way to write this is a filter over **emails** which uses the blacklisted servers as a broadcast variable (in Spark syntax):

```
val bl = sc.broadcast(...) // broadcast blacklist
emails.filter(e => bl.value.exists(_.ip == e.ip))
```

Hardcoding this dataflow, however, might become a bottleneck if the number of worker nodes or the size of the blacklist increases. In Emma, the programmer can still write the very same expression:

```
// as for-comprehension
for (e <- emails; if bl.exists(_.ip == e.ip) yield e
// or directly in de-sugared form
emails.withFilter(e => bl.exists(_.ip == e.ip))
```

The **exists**-unnesting rule presented in Section 4.1 will ensure that the expression is flattened and seen as a (logical) join (see Section 4.3). The dataflow compiler can then decide whether to use a broadcast or a re-partition strategy in order to evaluate the join node at runtime. Another instance of this optimization (TPC-H Query 4) is depicted and discussed in Appendix A.

4.2.2 Fold-Group Fusion

An algebraically grounded view of all dataflows in an input program allows us to derive and implement non-trivial logical optimizations. The most important of these optimizations is *fold-group fusion*. Candidates for rewrites are **groupBy** terms where the group **values** are exclusively used as **fold** inputs. When the optimization is triggered, it replaces the **groupBy** with an **aggBy**, which fuses the group construction performed by the **groupBy** with the subsequent **fold** applications on the constructed group values. In terms of the APIs discussed in Section 1, this equals to replacing a **groupBy** with an equivalent **reduceByKey** when possible.

Detecting Candidates for Rewriting. The rewriting process goes as follows. First, we iterate over all comprehensions where at least one generator binds to a **groupBy** expression. In the k-means example, such a comprehension is only found for the (inlined) **newCtrds** expression at line 21:

$$\llbracket\, t \mid clr \leftarrow \llbracket \ldots \rrbracket.\texttt{groupBy}(x \mapsto x.cid) \,\rrbracket^{\mathsf{Bag}}$$

We then look into the AST subtree at t and check whether all occurrences of the group **values** are enclosed in a comprehended term. If this is the case, and **values** appears only in generators for comprehended **fold** terms, the optimization can be triggered. This is the case in the **newCtrds** expression, as **clr.values** occurs only in the following two comprehensions:

$$\llbracket\, x.p.pos \mid x \leftarrow clr.values \,\rrbracket^{\texttt{fold}(zeros, x\mapsto x, (x,y)\mapsto x+y)} \qquad (\textsc{Sum})$$

$$\llbracket\, x.p.pos \mid x \leftarrow clr.values \,\rrbracket^{\texttt{fold}(0, x\mapsto 1, (x,y)\mapsto x+y)} \qquad (\textsc{Cnt})$$

The rewrite triggered by the optimization is derived from two generic algebraic equivalences – *banana split* and *fold-build fusion* that hold for *any* ADT and its associated **folds**.

Banana Split. The first law that enables fold-group fusion essentially says that a tuple of folds can be rewritten as a fold over tuples [28]. Generally speaking, the law generalizes the machinery behind loop fusion for arbitrary recursion schemes.[2] For example, the two fold comprehensions above can be substituted by a single comprehension which folds pairs by pairwise application of the original functions:

$$f_1 \times f_2 : (x_1, x_2) \mapsto (f_1(x_1), f_2(x_2))$$

The (e_1, s_1, u_1) triple thereby comes from Sum and operates on the first component, while (e_2, s_2, u_2) comes from Cnt and operates on the second:

$$\llbracket\, x.p.pos \mid x \leftarrow clr.values \,\rrbracket^{\texttt{fold}(e_1\times e_2, s_1\times s_2, u_1\times u_2)}$$

Fold-Build Fusion. The second law enables a rewrite which in functional programming languages is commonly known as *deforestation*. Intuitively, the law states that an operation that *constructs* an ADT value by means of its initial algebra constructors can be *fused together* with an operation that *computes* something from that value by means of structural recursion. For *fold-group fusion*, we want to fuse the **groupBy** operator, which constructs the group **values** with the single **fold** derived after the *banana-split* application. To illustrate the mechanics of the rewrite, we need to take a conceptual view on **groupBy** as a comprehension:

$$\llbracket\, (k(x), \llbracket\, y \mid y \leftarrow xs,\ k(x) = k(y) \,\rrbracket^{\mathsf{Bag}}) \mid x \leftarrow xs \,\rrbracket^{\mathsf{Set}}$$

With this conceptual view, the **groupBy** operator is interpreted as two nested comprehensions. The outer comprehension constructs the set of groups, each identified with a group key $k(x)$. The inner comprehension selects all xs elements that share the current group key to construct the group **values**. For fold-group fusion, we conceptually replace the **Bag** algebra that constructs the **values** with the **fold** algebra which evaluates them:

$$\llbracket\, (k(x), \llbracket\, y \mid y \leftarrow xs,\ k(x) = k(y) \,\rrbracket^{\texttt{fold}(\ldots)}) \mid x \leftarrow xs \,\rrbracket^{\mathsf{Set}}$$

Rewriting. In practice, however, we realize fold-group fusion by substituting the **groupBy** operator with an **aggBy** operator. Instead of a $(key, values)$ pair, this operator produces a tuple of aggregates $(key, a_1, \ldots, a_n)$. In our running example, a_1 will hold the result of Sum, and a_2 the result of Cnt:

$$\begin{aligned} \llbracket\, t' \mid clr \leftarrow \llbracket \ldots \rrbracket.\texttt{aggBy}(x \mapsto x.cid,& \\ \mapsto (zeros,\ 0),& \\ x \mapsto (x,\ 1),& \\ (x,y) \mapsto (x_1 + y_1,\ x_2 + y_2)) \,\rrbracket^{\mathsf{Bag}}& \end{aligned}$$

[2] Loop fusion can be seen as the law instance when the underlying ADT is a collection in AlgBag-Ins representation.

$$\llbracket\, \texttt{f}\ x \mid x \leftarrow xs \,\rrbracket \ \Rightarrow \quad \texttt{map}\ \texttt{f}\ xs \qquad (\textsc{Map})$$

$$\texttt{flatten}\llbracket\, \texttt{f}\ x \mid x \leftarrow xs \,\rrbracket \ \Rightarrow \quad \texttt{flatMap}\ \texttt{f}\ xs \qquad (\textsc{FlatMap})$$

$$\llbracket\, t \mid qs,\ x \leftarrow xs,\ qs_1,\ \texttt{p}\ x,\ qs_2 \,\rrbracket \ \Rightarrow \quad \llbracket\, t \mid qs,\ x \leftarrow \texttt{filter}\ \texttt{p}\ xs,\ qs_1,\ qs_2 \,\rrbracket \qquad (\textsc{Filter})$$

$$\begin{aligned}&\llbracket\, t \mid qs, x \leftarrow xs,\ y \leftarrow ys,\\ &\quad qs_1,\ \texttt{k}_1\ x = \texttt{k}_2\ y,\ qs_2 \,\rrbracket \ \Rightarrow\\ &\quad\quad \llbracket\, t[v.x/x][v.y/y] \mid qs,\ v \leftarrow \texttt{join}\ \texttt{k}_2\ \texttt{k}_2\ xs\ ys,\\ &\quad\quad\quad qs_1[v.x/x][v.y/y],\ qs_2[v.x/x][v.y/y] \,\rrbracket\end{aligned} \qquad (\textsc{EqJoin})$$

$$\begin{aligned}&\llbracket\, t \mid qs, x \leftarrow xs,\ y \leftarrow ys,\ qs_1 \,\rrbracket \ \Rightarrow\\ &\quad\quad \llbracket\, t[v.x/x][v.y/y] \mid qs,\ v \leftarrow \texttt{cross}\ xs\ ys,\\ &\quad\quad\quad qs_1[v.x/x][v.y/y] \,\rrbracket\end{aligned} \qquad (\textsc{Cross})$$

Figure 2: Combinator rules for dataflow construction.

To compensate for the rewrite in the original head expression t, we also substitute the terms associated with the Sum and Cnt comprehensions with $\texttt{a}_1$ and $\texttt{a}_2$, correspondingly, and obtain a new head t'.

4.3 Code Generation

As a result of the logical optimizations from Section 4.2, we obtain a modified AST and comprehension view. In the last phase of the compilation process, we use those to construct an AST that delegates the execution of the comprehended terms to a parallel runtime. The process is realized in two steps: (*i*) generating abstract dataflows for the comprehensions at the top-most level, and (*ii*) injecting glue code to evaluate the abstract dataflow plans just-in-time and embed the results in the surrounding driver AST.

4.3.1 From Comprehensions to Dataflows

In the first step, we use the comprehension views to derive abstract dataflow expressions close to the form expected by the parallel runtime. Similar to the normalization described in Section 4.1, this operation is also rule-based and follows rather closely the approach suggested by Grust in [20, 22]. In this approach, each rule matches two or more elements from a comprehension and replaces them with a single, closed form expression called a *combinator*. The rewrite continues until all comprehension elements are translated, and returns a combinator tree as an end result.

Combinators as Target API. In the original work, the author alludes to the correspondence between the inferred combinators and the logical operators supported by the targeted query engine. A similar correspondence can be drawn to the higher-order functions supported by the targeted parallel dataflow engines. Consequently, the resulting combinator trees effectively represent an abstract version of the parallel dataflows we want to execute.

Rewrite Rules. The set of combinator rules for dataflow construction that we use is listed in Figure 2. The rewrite process implements the state machine depicted in Figure 3a. At each state, the machine tries to apply any of its associated rewrite rules. If a match is found, the state transitions along the solid edge; if not, it follows the dashed edge. This heuristic strategy ensures that filters are pushed down as much as possible in the constructed dataflow tree, followed by joins, and then by cross products.

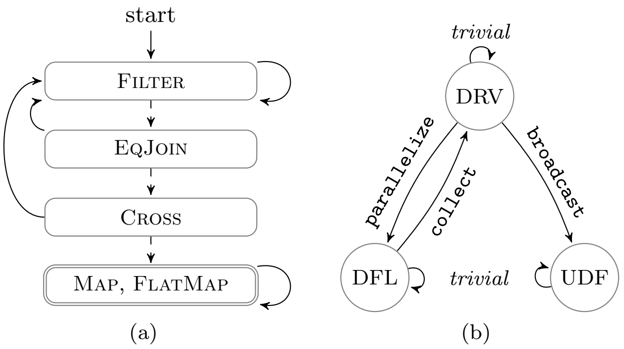


Figure 3: (a) Combinator rewrite process; (b) Data motion patterns in parallel dataflow engines

In principle, all but the I/O and the type conversion operators listed in Listing 3 can be defined in terms of comprehensions (we already saw this for `groupBy` in Section 4.2.2). This, however, appears to have more downsides than benefits. On the one hand, it forces us to extend the set of combinator rules in order to re-discover the comprehended operators. On the other hand, due to their particular structure, adding these comprehensions to the intermediate representation does not increase the potential for logical optimizations. To simplify the translation process, at the moment we just substitute non-comprehended operators with equivalent combinator expressions.

Example. The abstract dataflow that computes `newCtrds` at line 21 looks as follows:

$$\begin{aligned}&\mathit{points}\\ &\quad .\texttt{map}(p \mapsto \{\\ &\quad\quad \mathit{Solution}(\mathit{ctrds.minBy}(\mathit{distanceTo}(p)).\mathit{get.id},\ p)\\ &\quad \})\\ &\quad .\texttt{aggBy}(x \mapsto x.\mathit{cid},\ (e_1 \times e_2, s_1 \times s_2, u_1 \times u_2))\\ &\quad .\texttt{map}(\mathit{clr} \mapsto \{\\ &\quad\quad \mathit{Point}(\mathit{clr.key},\ \mathit{clr.a}_1/\mathit{clr.a}_2)\\ &\quad \})\end{aligned}$$

Since the combinators in the derived abstract dataflows have a corresponding operator in each runtime, generating a concrete dataflow boils down to simple node substitution in the abstract dataflow tree. However, most dataflow APIs allow the user to give additional physical hints (e.g. with respect to partitioning, caching, or algorithm selection), and ideally these should also be set as part of the translation. Since such physical aspects depend on the execution environment, we trigger the actual dataflow generation just-in-time at runtime. To do this, we emit Scala code that assembles an abstract dataflow with reified UDFs so we can access their ASTs at runtime. The dataflow is then wrapped in a thunk object which represents the actual result (see the next section) and is compiled and evaluated lazily at runtime.

4.3.2 Transparent Data Motion

With the steps described so far, we already can generate and execute dataflows that read and write data from a storage layer and are independent from the enclosing code. Most algorithms for complex data analysis, however, have more complex data dependencies.

Conceptually, data can be produced and consumed by three different agents – the driver code (DRV), the parallel dataflows (DFL), and the dataflow UDFs. The diagram in Figure 3b illustrates the dependencies that a typical dataflow engine supports and the API primitives that realize them. Connecting dataflow sources and sinks with code that runs in the driver is facilitated in Spark by `parallelize` and `collect` calls, correspondingly. In addition, driver variables can be made available to all dataflow UDFs with a `broadcast` primitive. Equivalents for `parallelize`, `broadcast` and `collect` can be also found in Flink. Data dependencies within the same agent are trivial and do not require special treatment during compilation.

Driver to UDFs. To integrate data motion between the driver and the UDFs, we check the functions in the dataflows generated in Section 4.3.1 for unbound variables. For each unbound variable, we inject a corresponding `broadcast` call and modify the UDF code accordingly. An example of such unbound variable is the `ctrds` used in the first mapper for the `newCtrds` dataflow.

Driver to Dataflows. To integrate data motion between the driver and the dataflows, we wrap the dataflows generated in Section 4.3.1 in special objects called *thunks* [16]. A thunk object of type Thunk[A] wraps an abstract dataflow that results in a value of type A. Traditionally, a thunk has just one method – `force`, which forces execution, memoizes the result, and returns it to the client. To adapt the concept to our needs, however, we also add a `dataflow` method, as in some situations we want to access the head of the abstract dataflow instead of its actual result.

We define both `force` and `dataflow` as *implicit converters*. The Scala type checker then automatically injects them in expressions which expect a parameter of type A or DFL[A], but receive an argument of type Thunk[A] instead.

4.4 Physical Optimizations

The combination of a full view over the program AST at compile-time with a just-in-time compilation approach for the constructed abstract dataflows opens potential for a range of physical optimizations. In the following paragraphs, we briefly present two such optimizations together with naive strategies to apply them. Refining the pysical aspects of the language compiler (e.g. with a cost-based optimizer) is one our areas for future research.

Caching. In Spark and Flink, `RDD` and `DataSet` transformations are lazy, and the decision upon materialization of intermediate results is left to the client. An open question in this situation is when to do this and when not. As an aggresive heuristic strategy, at the moment we force the evaluation and caching of dataflow results that are referenced more than once (e.g. inside a loop or within multiple branches) in the compiled algorithm.

Partition Pulling. Execution of certain operators (e.g. `equi-join`, `group`) in a parallel setting usually enforces hash partitioning on their inputs. Partitionings that can be reused by a certain dataflow (e.g. on a join or group key) can be spotted by Emma and enforced earlier in the pipeline. More precisely the view over the control flow structure of the driver program allows us to detect and pull enforced partitioning behind control flow barriers in order to minimize the number of overall shuffles in an algorithm. The heuristic we suggest at the moment is to (*i*) compute the sets of interesting partitionings for each dataflow result based on its occurence in other dataflow inputs, and (*ii*) enforce a partitioning at the producer site based on a weighted scheme that prefers consumers occuring within a loop structure. A cost-based approach is likely to produce better results in this situation, but goes beyond the scope of our current work.

Optimization Algorithm	Unnesting	Group Fusion	Cache	Partition Pulling
Data-Prl. Workflow	✓	×	✓	✓
k-means	×	✓	✓	×
PageRank	×	✓	✓	×
TPC-H Q1	×	✓	×	×
TPC-H Q4	✓	✓	×	×

Table 1: Programs presented in the experiments and optimizations that apply to them (marked by ✓).

Listing 5: Selecting a Spam Email Classifier in Emma

```
val emails = read(/* from file */).map(extractFeatures)
val blacklist = read(/* from file */)
var minHits = -1L
var minClassifier = null

// for each different classifier
for (c <- SpamClassifiers) {
  // find out which of the emails are spam
  val nonSpamEmails = for (
      email <- emails;
      if not c.isSpam(email))
    yield email

  // non-spam emails coming from a blasklisted server
  val nonSpamFromBlServer = for (
      email <- nonSpamEmails;
      if blacklist.exists(_.ip == email.ip))
    yield email

  // track the classifier which leads to minimum count
  if (nonSpamFromBlServer.count() < minHits) {
    minHits = nonSpamFromBlServer.count()
    minClassifier = c
  }
}
```

5. EVALUATION

In this section we evaluate the effectiveness of the generated code that results from Emma's compiler pipeline. We consider a set of programs coming from different domains, namely: a data-parallel workflow (Section 5.1), iterative dataflows and relational queries (Section 5.2). Table 1 shows a list of the implemented algorithms and the optimizations that apply to each of them.

Experimental Setup. We implemented a prototype of the compiler pipeline presented in Section 4 on Scala 2.11 using Scala Macros. We ran our experiments on a cluster of 40 machines (8 cores, 16GB RAM) running Flink v0.8.0[3], and Spark v1.2.0[4]. Every experiment was executed 7 times – we report the median execution time.

5.1 Data-Parallel Workflow

In order to show the effect of the optimizations described in sections 4.2 and 4.4 we devised an advanced workflow that includes UDFs, a sequential loop, and an `if` statement. The worklfow's inputs are: (*i*) a set of trained spam email classifiers; (*ii*) a set of emails, and; (*iii*) a mail server blacklist. The code for this workflow can be found in Listing 5.

[3] Available at http : //flink.apache.org/

[4] Available at http : //spark.apache.org/

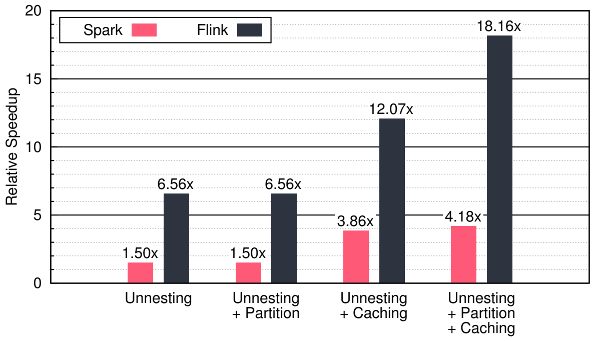


Figure 4: Effect of the different optimizations to the runtime of the Data-Parallel Workflow.

The workflow goes as follows: first, it reads a set of emails and applies a feature extractor on each email (line 1). It then reads a list of blacklisted mail servers (line 2). The loop at lines 7-25 applies different classifiers (lines 9-12) and finds the one that produces the least amount of non-spam emails sent from blacklisted servers (lines 15-18).

Applicable Optimizations. This program is subject to all optimizations listed in Table 1, except for Fold-Group-Fusion. The applicable optimizations are detailed below:
– *Unnesting.* As shown in Section 4.2.1, flattening the `blacklist.exists` allows the use of a re-partition join on the `ip` values instead of a filter with a broadcasted `blacklist`.
– *Caching.* The contents of `emails` are referenced, but not modified within the loop at lines 7-25, so the expression at line 1 can be cached based on the heuristic discussed in Section 4.4.
– *Partition Pulling.* The `nonSpamEmails` expression preserves the partitioning of `emails`. Enforcing a partitioning on `ip` at both line 1 and 2, will propagate to line 15; if the results are cached (see "caching" above) before entering the loop, the shuffle step can be executed only once rather than once per classifier.

Dataset. We generated a synthetic dataset comprising a blacklist and a set of emails. The blacklist file contains 100.000 blacklisted IPs along with information about each server (2GBs). The emails file contains 1M emails with various fields (mailserver IP, subject, body, etc.). The average size of an email is 100KBs resulting to a total size of 100GBs.

Effect of Optimizations. Figure 4 depicts the speedup of the workflow's execution time when a set of optimizations apply, relative to a baseline case with no optimizations. In the baseline case, Emma does not perform unnesting. As a result, the `blacklist` is broadcasted to all nodes (40 nodes × 2GBs), in order to execute a broadcast join, resulting in very large data exchange. When unnesting is applied (leftmost bars in Figure 4), the `blacklist` is instead partitioned, and a repartition join with `nonSpamEmails` takes place. The effect of unnesting alone accounts for a speedup of 1.5x in Spark and 6.56x in Flink. The difference in the speedup between Flink and Spark is due to specifics in Flink's current handling of broadcast variables.

Applying partition pulling (second set of bars from the left) in addition to unnesting does not bring any benefit. This is explained by that fact that, without caching, the databags are evaluated lazily in every iteration. Since partitioning is going to be enforced either way by the join, forcefully partitioning before the loop will have no effect. However, when we apply caching and partition pulling at the same time, both `blacklist` and `emails` can be partitioned on the `ip` field *before* they are cached (out of the loop). In this case (rightmost bars in Figure 4), the cost of the shuffle is paid only once and amortized over all iterations, causing a speedup of 18.16x for Flink and 4.18x for Spark. Note that this effect comes from the synergy of partitioning *and* caching; caching alone achieves a speedup of 12.07x in Flink and 3.86x in Spark (third set of bars) due to the amortization of the `extractFeatures` UDF (now executed only once).

The above observations showcase the potential of our proposed optimizations. However, the choice of the partitioning scheme and the best point to enforce it, as well as decisions regarding caching cannot be reliably handled by heuristics; the right choice relies on runtime statistics and a cost model.

5.2 Other Algorithms

In this section we show the effect of our optimizations on two iterative algorithms, namely: k-means, and PageRank as well as two TPC-H queries (Q1 and Q4). The code for k-means has been presented earlier (Listing 4), while the code for the other algorithms can be found in Appendix A. In Table 1 we can see that Fold-Group Fusion applies to all the above algorithms and queries, while caching applies only to the two iterative algorithms.

Datasets. For k-means we used 3 random fixed centers and 1.6B points (48GBs). For PageRank we used a Twitter Follower Graph [12] (23GBs, ~2B edges). For TPC-H we used the TPC-H dataset with a scaling factor of 50 and 100.

Iterative Algorithms. We ran the two algorithms without Fold-Group Fusion but they failed to finish within a timeout of one hour. We then applied Fold-Group Fusion and ran the algorithms with and without caching. Caching on Spark resulted into a speedup of 1.52x on k-means and 3.13x on PageRank. PageRank was sped up more than k-means since PageRank stores the vertices and their ranks already partitioned by the vertex ID in-memory in a form that is ready to be consumed by the next iteration. K-means merely caches the set of points that do not need any repartitioning, and thus, only the cost of reading the points from HDFS is saved.

Interestingly, in this experiment Flink did not show significant improvement by the use of caching: the current version of Flink does not provide a means to cache results in-memory, thus Emma caches intermediate results on HDFS. As a result, the benefits of caching are eliminated by the cost of the additional I/O.

TPC-H Queries. Q1 is subject to Fold-Group Fusion while Q4 is subject to both logical optimizations. As in the previous case, without the logical optimizations, none of the queries was executed within the limit of one hour. With logical optimizations enabled, both queries managed to finish their execution within 10 minutes (466s for Q1 on Spark and 240s on Flink; 577s for Q4 on Spark and 569s for Flink).

5.3 Conclusions from the Experiments

In our experiments we showed that the various optimizations enabled by Emma can have a substantial impact on the runtime of an algorithm. Omitting the logical optimizations can be the reason for an algorithm to not execute at all, either because it is too slow, or because of memory issues.

Caching was shown to positively affect the runtime of iterative algorithms. As a matter of fact, under the assumption that enough memory is available in the cluster machines, an optimizer can safely make the heuristic decision to use caching whenever possible, as a way to reduce network IO and CPU cost (less shuffles). Moreover, keeping the intermediate results partitioned can further boost the performance.

6. RELATED WORK

General purpose APIs that programmatically construct dataflows (or, data pipelines) such as the Spark [35] or Flink [6] API, Cascading [1], Scalding [3] and Scoobi [4], were discussed in detail in previous sections. In the current section we present the rest of the related work.

Deeply Embedded Query Languages. LINQ [27] allows SQL-like querying constructs to be integrated in C#. Similarly, Ferry [21] is a comprehensions-based programming language that facilitates database-supported execution of entire programs. To be evaluated, LINQ and Ferry programs are both mapped into an intermediate algebraic form suitable for execution on SQL:1999-capable relational database systems. Similar to Ferry and LINQ, Emma is a comprehensions-based language, but does not only focus on SQL/RDBMS execution. Instead, it targets data-parallel execution on general-purpose distributed execution engines. The code generated by Emma is optimized Scala programs that make use of external systems APIs. Moreover, Emma is able to analyze comprehensions, enabling an important class of shared-nothing optimizations (e.g., group fusion).

The Delite project [13] provides a compiler framework for domain-specific languages. Delite's core intermediate representation is based on primitives similar to the higher-order functions used by Flink and Spark. Based on this representation, Delite generates optimized executable kernels for different hardware devices and schedules them from a centralized "interpreter" runtime. Emma proposes the use of monad comprehensions based on folds in union representation as alternative intermediate representation for data-parallel collection expressions. We believe that monad comprehensions provide a better formal ground for reasoning and applying non-trivial algebraic optimizations.

Jet [5] is a data-parallel processing language that, is deeply embedded in Scala. Jet uses a generative programming approach called *lightweight modular staging* (LMS) [31] to analyze the structure of user programs and apply optimizations like projection insertion, code motion, operation fusion, etc.

Languages for Cluster Data-Parallel Computing. DryadLINQ [34] allows to execute LINQ expressions on Dryad [24]. It requires a dataflow to be expressed via a *single* SQL-like statement, while Emma allows for more expressive programs that are analyzed and optimized holistically - in Emma dataflows are inferred in the user program.

Pig [30] and Hive [32] are SQL-inspired languages that are restricted in order to achieve relational optimizations, while UDFs are specified as separate Java programs. This makes their implementation cumbersome.

The SCOPE [14] scripting language follows a SQL-like syntax that can be extended with C# for custom UDFs. Several optimizations apply to SCOPE scripts [23] ranging from traditional distributed database optimizations (partitioning, join order, column reductions, etc.) to lower level code optimizations, such as code motion.

To the best of our knowledge, Emma is the first embedded language for large scale data-parallel processing that utilizes a holistic view of the user code at compile time to enable implicit parallelism. Emma hides low level API primitives behind a declarative, ubiquitous `DataBag` abstraction and performs an important algebraic optimizations previously unattained by similar languages to ensure high performance. Emma translates a program into a driver and a set of parallel dataflows that can be executed on multiple backends.

Optimizing Group Aggregations. There is a large body of work studying aggregations in the parallel/distributed literature [15, 19]. Parallel databases such as DB2, Gamma, Volcano and Oracle support pre-aggregation techniques for SQL. However, when aggregations involve more complex user defined functions and complex data types, optimization opportunities are greatly reduced. To solve this issue, [33] provides programmers an easier interface to program optimizable user defined aggregate functions. Moreover, [33] can produce pre-aggregators (or, combiners) when a provided reducer function is declared as decomposable (or associative-decomposable). The group fusion method we describe in Section 4.2.2 does not require users to annotate their code neither to follow any pre-defined template for generating pre-aggregators; instead, it generalizes to all aggregates that are programmed using folds.

More recently, Murray et. al. [29] discussed how group fusion can be applied in the context of code generation for LINQ. Their view of aggregates, is based on ALGBAG-INS and requires additional "homomorphy" constraints on the aggregates. Emma, by using ALGBAG-UNION, ensures that this condition is always fulfilled, while applying banana-split allows to fuse multiple folds simultaneously.

7. CONCLUSIONS AND FUTURE WORK

Current data-parallel analysis languages and APIs either lack declarativity or are not expressive enough to capture the complexity of today's data-parallel analysis requirements. To this end, we presented Emma – a language deeply embedded in Scala, that provides implicit parallelism through declarative dataflows. We presented an intermediate representation and a novel compiler pipeline based on monad comprehensions over bags in union representation. Emma programs are compiled down to driver programs that utilize APIs of existing data-parallel execution engines. Finally, we presented a set of experiments that demonstrate that Emma can provide declarativity without sacrificing performance.

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 (e.g., join order optimization). To this end, we are currently working on an optimizer that will generate the dataflows just in time. We are investigating the propagation of properties across different dataflows and the possibility to switch deployment strategies (e.g., lazy loop unrolling vs. native/persistent iterations) at runtime.

Moreover, domain-specific abstractions can be easily integrated on top of the `DataBag` API through Scala macros. We are developing linear algebra and graph processing APIs on top of the `DataBag` API.

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 (funding mark 01IS14013A) and by the EIT ICT Labs (EUROPA 2014 project).

8. REFERENCES

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

[2] Scala language reference. `http://www.scala-lang.org/files/archive/spec/2.11/`.

[3] Scalding Project. `http://github.com/twitter/scalding`.

[4] Scoobi Project. `http://github.com/nicta/scoobi`.

[5] S. Ackermann, V. Jovanovic, T. Rompf, and M. Odersky. Jet: An embedded dsl for high performance big data processing. In *BigData*, 2012.

[6] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinlaender, M. J. Sax, S. Schelter, M. Hoeger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *VLDB Journal*, 2014.

[7] 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*, 2011.

[8] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. *PVLDB*, 2010.

[9] P. Buneman, L. Libkin, D. Suciu, V. Tannen, and L. Wong. Comprehension Syntax. *SIGMOD Record*, 1994.

[10] P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theor. Comput. Sci.*, 1995.

[11] E. Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, page 3. ACM, 2013.

[12] M. Cha, H. Haddadi, F. Benevenuto, and P. K. Gummadi. Measuring user influence in twitter: The million follower fallacy. *ICWSM*, 2010.

[13] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *ACM SIGPLAN Notices*, 2011.

[14] 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.

[15] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *VLDB*, 1994.

[16] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: being lazy is a virtue (when issuing database queries). In C. E. Dyreson, F. Li, and M. T. Özsu, editors, *ACM SIGMOD*, 2014.

[17] H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1: Equations and initial semantics.* Springer Publishing Company, Incorporated, 2011.

[18] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning Fast Iterative Data Flows. *PVLDB*, 2012.

[19] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 1993.

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

[21] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber. Ferry: Database-supported program execution. In *SIGMOD.* ACM, 2009.

[22] T. Grust and M. H. Scholl. How to comprehend queries functionally. *J. Intell. Inf. Syst.*, 1999.

[23] Z. Guo, X. Fan, R. Chen, J. Zhang, H. Zhou, S. McDirmid, C. Liu, W. Lin, J. Zhou, and L. Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI*, pages 121–133, 2012.

[24] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS*, 2007.

[25] W. Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems (TODS)*, 7(3):443–469, 1982.

[26] J. Lambek. A fixpoint theorem for complete categories. *Mathematische Zeitschrift*, 1968.

[27] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the. net framework. In *ACM SIGMOD*, 2006.

[28] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. pages 124–144. Springer-Verlag, 1991.

[29] D. G. Murray, M. Isard, and Y. Yu. Steno: automatic optimization of declarative queries. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 121–131. ACM, 2011.

[30] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ACM SIGMOD, 2008.

[31] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In *Acm Sigplan Notices.* ACM, 2010.

[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *VLDB*, 2009.

[33] Y. Yu, P. K. Gunda, and M. Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *ACM SIGOPS*, 2009.

[34] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[35] 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.

APPENDIX

A. EXAMPLE CODE

In the following, we provide Emma code for four additional algorithms (two graph algorithms and two relational queries) and provide comments that should highlight some of the unique features of the language based on concrete code examples.

A.1 Graph Algorithms

A.1.1 PageRank

The algorithm is roughly divided in two parts for each iteration. (1) In lines 3-9 we calculate the current `rank` of each vertex by joining each `rank` entry (retrieved from a `StatefulBag`) with its `vertices` adjacency list. For each pair, the comprehension yields a `RankMessage` which contains the current vertex `rank` divided by the number of its neighbors. (2) We then group the messages by `vertex id` (line 11) and calculate the new rank by summing up the `ranks` (line 12) and applying the rank formula in line 13. Finally, we point-wise update the `ranks` state (see StatefulBags in Programming Abstractions 3.1). In this variant of PageRank we iterate for a fixed number of iterations, although in principle a termination criterion based on global rank change can used as well.

Listing 6: Page Rank in Emma

```
1  var iter = 0
2  while (iter < maxIterations) {
3    val messages = for (
4      p <- ranks.bag();
5      v <- vertices; n <- v.neighbors;
6      if p.id == v.vertex) yield {
7        RankMessage(n, p.rank / v.neighbors.count())
8      }
9
10   val updates = for (
11     g <- messages.groupBy(_.vertex)) yield {
12       val inRanks = g.values.map(_.rank).sum()
13       val newRank = (1 - DF)/numPages + DF * inRanks
14       VertexWithRank(g.key, newRank)
15   }
16
17   ranks.update(updates)((s, u) =>
18       Some(s.copy(rank = u.rank)))
19
20   iter += 1
21 }
```

A.1.2 Connected Components

In lines 1-3 we map each `vertex` to the `State` object consisting of `(id, neighbors, component)` triple and save the result as a `StatefulBag`. We update the state iteratively in a semi-naive manner while the changed `delta` is not empty (line 5). Initially, the `delta` contains all vertices. In lines 6-7, we create `Messages` for all neighbors of the vertices in the current `delta`. We then find the highest vertex id by neighbor (lines 9-10) and save it as an `update` object. The updates are then compared point-wise against their state counterparts in order to determine whether an update is actually required (line 13). If this is the case, the new version of the state at this particular point is returned in the `Some(...)` expression at line 14. The delta is then transparently merged into the current state and forwarded back to the client, where a stateless (`DataBag`) version of it is assigned to the `delta` variable (see 3.1).

Listing 7: Connected Components in Emma

```
1  var delta = for(v <- vertices) yield
2    State(v.id, v.neighborIDs, v.id)
3  val state = stateful[State, VID](delta)
4
5  while (not delta.empty()) {
6    val msgs = for(s <- delta; n <- s.neighborIDs)
7      yield Message(n, s.component)
8
9    val updates = for(g <- msgs.groupBy(_.receiver))
10     yield Updt(g.key, g.values.map(_.component).max())
11
12   delta = state.update(updates)((s, u) =>
13     if (u.component > s.component)
14       Some(s.copy(component = u.component))
15     else
16       None)
17 }
```

A.2 TPC-H Queries

A.2.1 TPC-H Query 1

We read and filter `lineitem` by the specified date in lines 2-5 before grouping and performing the final aggregation in lines 8-32. We thereby first group `lineitem` by `returnFlag` and `lineStatus` (line 9) and then specify the aggregate expressions in the actual comprehension head (lines 12-19). Note that in other dataflow APIs the programmer would have to perform the Fold-Group Fusion rewrite manually in order to achieve the same performance. This entails (*i*) applying the banana split law in order to treat the six folds at lines 12-19 as a composite fold over a tuple with six components, and (*ii*) combining the `e`, `s`, and `u` arguments of this fold in a "pre-processing" `map` and `reduceByKey`, which correspondingly produce and merge the 6-tuples encoding the result of the fused folds (see also lines 5-6 in Listing 1).

Listing 8: TPC-H Query 1 in Emma

```
1  // compute join part of the query
2  val l = for (
3    l <- read(...LineItem...);
4    if l.shipDate <= "1996-12-01")
5  yield l
6
7  // aggregate and compute the final result
8  val r = for (
9    g <- l.groupBy(l => GrpKey(l.returnFlag, l.lineStatus)))
10 yield {
11   // compute base aggregates
12   val sumQty = g.values.map(_.quantity).sum();
13   val sumBasePrice = g.values.map(_.extendedPrice).sum()
14   val sumDiscPrice = g.values.map(l =>
15     l.extendedPrice * (1 - l.discount)).sum()
16   val sumCharge = g.values.map(l =>
17     l.extendedPrice*(1 - l.discount)*(1 + l.tax)).sum()
18   val countOrder = g.values.count()
19   val sumDiscount = g.values.map(_.discount).sum()
20   // compute result
21   Result(
22     g.key.returnFlag,
23     g.key.lineStatus,
24     sumQty,
25     sumBasePrice,
26     sumDiscPrice,
27     sumCharge,
28     avgQty = sumQty / countOrder,
29     avgPrice = sumBasePrice / countOrder,
30     avgDisc = sumDiscount / countOrder,
31     countOrder)
32 }
```

A.2.2 TPC-H Query 4

In lines 1-2 we read from the file system. Lines 4-13 represent the **where** part of the corresponding SQL statement. For every tuple in **orders**, we filter by the date interval (lines 6-7) and apply the **exists** quantifier. Note that we retain the same syntactic level of declarativity as SQL, while the unnesting rule (Section 3.1) allows us to avoid hard coding the evaluation strategy for the **exists**. In other dataflow APIs, the user would have to express the **exists** clause either by a broadcast variable or a join on the filtered **lineitem** tuples. In line 13 we return the order priority for all matching tuples. Finally, we group the resulting tuples by their priority and count them (lines 16-18).

Listing 9: TPC-H Query 4 in Emma

```
1  val lineitems = read(...LineItem...)
2  val orders = read(...Orders...)
3
4  val join = for (
5    o <- orders
6    if o.orderDate >=  dateMin &&
7       o.orderDate <   dateMax &&
8       lineitems.exists( li =>
9           li.orderKey  == o.orderKey &&
10          li.commitDate < li.receiptDate
11      )
12   )
13   yield Join(o.orderPriority, 1)
14
15 // aggregate and compute the final result
16 val rslt = for (
17   g <- join.groupBy(x => x.orderPriority)) yield
18     println(g.key, g.values.count())
```

B. EXTRA EXPERIMENTS

B.1 Effect of Fold-Group Fusion

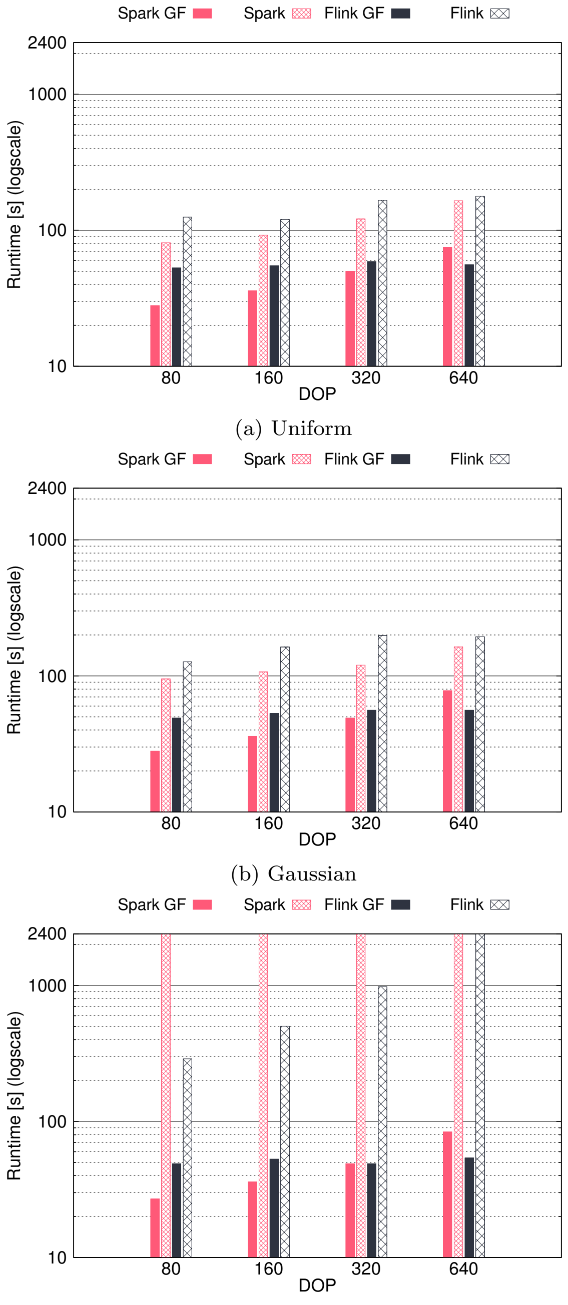
In this section we show a set of experiments that analyze the effect of the Fold-Group Fusion (GF) optimization that we introduced in Section 4.2.2. To this end, we run a group aggregation query over different degrees of parallelism (DOP, for short) and data distributions. To see the effect of GF, we instruct our optimizer to produce plans for Spark and Flink. For each of the systems, the optimizer outputs a plan that is optimized with GF and one that is not.

Dataset. We generated 3 synthetic datasets whose keys follow: (*i*) a uniform; (*ii*) a Gaussian; (*iii*) a Pareto distribution (assigning ~35% of all tuples on one key). Each tuple in the datasets consists of three fields: a *key* (integer), a *value* (integer), and a small random *payload* (3-10 unicode characters). We provide each spawned execution unit with 5M tuples (~125MBs). As a result, the size of the dataset increases proportionally to the DOP.

Query. We run the aggregation below for different DOPs:

```
for (g <- dataset.groupBy(_.key))
  yield (g.key, g.values.map(_.value).min())
```

Analysis. Figure 5 shows the running times for Flink and Spark with increasing DOP (80-640) for the three data distributions. A first observation is that fold-group fusion enables both execution engines to successfully compute the aggregation on all data distributions almost without any overhead. This was expected, as local, partial pre-aggregations take place on the mapper side and the reducers that execute the global aggregation are not overloaded – exactly one aggregated tuple per key is sent from each mapper to the respective reducer.



(a) Uniform

(b) Gaussian

(c) Pareto

Figure 5: Effect of fold-group fusion to the scalability of a group aggregation (min) in different data distributions.

When GF is not used, both engines on the Gaussian distribution dataset need slightly more time to compute the aggregate, while for the Pareto distribution, Spark fails to finish any aggregation within the given time limit (40 min). A last observation is that Flink scales linearly to the DOP in all cases where GF is used, while Spark exhibits a superlinear behavior.