PROTEUS
Scalable online machine learning for predictive analytics and real-time interactive visualization
687691

D3.7 Declarative language first prototype
Authors: Bonaventura Del Monte, Jeyhun Karimov, Alireza Rezaei Mahdiraji
Reviewer: Hamid Bouchachia

<table>
<thead>
<tr>
<th>Deliverable nature:</th>
<th>Demonstrator (D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dissemination level:</td>
<td>Public (PU)</td>
</tr>
<tr>
<td>Confidentiality</td>
<td></td>
</tr>
<tr>
<td>Contractual delivery date:</td>
<td>May 31\textsuperscript{th} 2017</td>
</tr>
<tr>
<td>Actual delivery date:</td>
<td>May 31\textsuperscript{th} 2017</td>
</tr>
<tr>
<td>Version:</td>
<td>1.2.1</td>
</tr>
<tr>
<td>Total number of pages:</td>
<td>30</td>
</tr>
<tr>
<td>Keywords:</td>
<td>Machine learning, Linear Algebra, Relational Algebra</td>
</tr>
</tbody>
</table>
Abstract

Advanced data analysis typically requires some form of pre-processing to extract and transform data before processing it with machine learning and statistical analysis techniques. Pre-processing pipelines are naturally expressed in dataflow APIs (e.g., MapReduce, Flink, etc.), while machine learning is expressed in linear algebra with iterations. Programmers therefore perform end-to-end data analysis utilizing multiple programming paradigms and systems. This impedance mismatch not only hinders productivity but also prevents optimization opportunities, such as sharing of physical data layouts (e.g., partitioning) and data structures among different parts of a data analysis program.

Our goal is twofold. First, we aim to alleviate the impedance mismatch by allowing programmers to author complete end-to-end programs in one engine-independent language that is automatically parallelized. Second, we intend to enable joint optimizations over both relational and linear algebra. To achieve these goals, we present the first prototype of Lara, a deeply embedded language in Scala that enables authoring scalable programs using two abstract data types (DataBag and Matrix) and control flow constructs. Programs written in Lara are compiled to an intermediate representation (IR) that enables optimizations across linear and relational algebra. IR is finally used to compile code for different execution engines.
Executive summary

In this deliverable, we show a first prototype of a declarative language, *Lara*, for machine learning requirements. Lara aims to allow programmers to define their machine learning algorithms in one engine-independent language and parallelize automatically. Moreover, it enables joint optimization over both relational and linear algebra.

While relational *domain-specific languages* (DSLs) are a good fit for Extract Transform Load (ETL) tasks, programming machine learning algorithms in those languages is cumbersome. To this end, DSLs offer linear algebra and control flow primitives suitable for expressing ML algorithms, but only provide limited, non-intuitive support for classic ETL tasks. This strict separation of programming paradigms introduces three fundamental problems, being *Impedance Mismatch*, *Inefficient Data Exchange* and *Loss of Optimization Potential*. To overcome these problems, we argue for the unification of relational and linear algebra in Lara.

The use-case for this language overlaps with one of main mission of PROTEUS project, i.e., to investigate and develop ready-to-use scalable online machine learning algorithms. As an example, the abundance of machinery sensors at ArcelorMittal, which produce a large amount of data, demands a processing system able to:

- predict degree of flatness of coil
- detect anomalies in the time series of the target variables

Such complex analytics can be expressed in *Lara* through a pipeline of transformations made of two parts: a ETL part written as a declarative program of transformations over *DataBags* and a ML part made of linear algebra operations. The underlying compiler of Lara takes the source code as input and determines the final physical execution strategy. The end-user is, therefore, relieved of taking care of any detail of the underlying processing system.

Our DSL is based around the concept of generic types. We propose a set of elementary generic types that model different structural aspects and represent both user-facing (e.g., matrix, bag) and engine-facing (e.g., partitioning) abstractions. Each type implies: (i) a set of structural constraints expressed as axioms, (ii) a set of structure-preserving transformations that operate element-wise and correspond to the functional notion of map.

Several holistic optimizations can be derived from the unified formal model and implemented under the assumption of a full view of the algorithm code, e.g., projection push down and filters. The former is based on the knowledge that fields are never accessed; the latter are applied on the originating *DataBags* of the matrices. In this first prototype, we present basic optimization, however we plan to introduce more sophisticated optimizations that come from a deeper analysis of a program code.

The first stable prototype of Lara is available at the following GitHub repository: [https://github.com/proteus-h2020/proteus-language](https://github.com/proteus-h2020/proteus-language). The new version of Lara will be pushed to the same repository in the future.
Advanced data analysis typically requires some form of pre-processing to extract and transform data before processing it with machine learning and statistical analysis techniques. Pre-processing pipelines are naturally expressed in dataflow APIs (e.g., MapReduce, Flink, etc.), while machine learning is expressed in linear algebra with iterations. Programmers therefore perform end-to-end data analysis utilizing multiple programming paradigms and systems. This impedance mismatch not only hinders productivity but also prevents optimization opportunities, such as sharing of physical data layouts (e.g., partitioning) and data structures among different parts of a data analysis program.

The goal of this work is twofold. First, it aims to alleviate the impedance mismatch by allowing programmers to author complete end-to-end programs in one engine-independent language that is automatically parallelized. Second, it aims to enable joint optimizations over both relational and linear algebra. To achieve this goal, we present the first
prototype of Lara, a deeply embedded language in Scala that enables authoring scalable programs using two abstract data types (DataBag and Matrix) and control flow constructs. Programs written in Lara are compiled to an intermediate representation (IR) that enables optimizations across linear and relational algebra. The IR is finally used to compile code for different execution engines.

Keywords
scalable machine learning, declarative language.

<table>
<thead>
<tr>
<th>Version Log</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Issue Date</strong></td>
</tr>
<tr>
<td>20.04.2017</td>
</tr>
<tr>
<td>16.05.2017</td>
</tr>
<tr>
<td>22.05.2017</td>
</tr>
<tr>
<td>22.05.2017</td>
</tr>
</tbody>
</table>
# Table of Contents

Executive summary ................................................................. 3  
Document Information .......................................................... 4  
List of figures and/or list of tables ........................................... 7  
Abbreviations ........................................................................... 8  
1 Introduction ............................................................................ 9  
   1.1 Motivating example ........................................................... 10  
   1.2 Discussion ....................................................................... 10  
   1.3 Document structure .......................................................... 11  
2 Related work ......................................................................... 12  
3 Language Foundations ............................................................ 13  
4 Optimizations ......................................................................... 15  
   4.1 Matrix Blocking Through Joins ......................................... 15  
   4.2 Row-wise Aggregation Pushdown ...................................... 16  
   4.3 Caching Through Iterations .............................................. 17  
5 Implementation Details ............................................................ 19  
   5.1 Emma: a Scala DSL for scalable data analysis ..................... 19  
      5.1.1 Compiler Infrastructure ............................................ 21  
      5.1.2 Emma Source Language ....................................... 23  
   5.2 Lara Internals .................................................................. 24  
      5.2.1 Matrix-Scalar Pointwise Operators ......................... 24  
      5.2.2 Matrix-Vector Pointwise Operators ...................... 24  
      5.2.3 Matrix-Matrix Pointwise Operators ...................... 24  
      5.2.4 Binary Matrix-Matrix and Matrix-Vector Operators .... 24  
      5.2.5 Matrix self-transformations Operators ................... 25  
6 Example Algorithms implemented in Lara ............................... 26  
   5.3.1 K-Means ..................................................................... 26  
   5.3.2 K-Means example ....................................................... 27  
   5.3.3 Linear Regression (with Gradient Descent optimization method) ........................................... 27  
   5.3.4 Linear Regression Example ........................................ 27  
5 Follow-up: online machine learning abstractions .................... 28  
6 Conclusion ............................................................................. 29  
References .............................................................................. 30
List of figures and/or list of tables

Figure 1. Two execution strategies for \texttt{toMatrix}(A \bowtie B) (colors represent different partitions). Above (a-c) Naive Approach. Below (d-f) Partition sharing. ................................................................. 16

Listing 1. Code snippet written in Lara ................................................................................. 10
Listing 2 K-Means implementation in Lara ........................................................................... 27
Listing 3 K-Means example .................................................................................................... 27
Listing 5 Linear Regression fitting example .......................................................................... 28
### Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>IR</td>
<td>Intermediate Representation</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain-Specific Language</td>
</tr>
<tr>
<td>ETL</td>
<td>Extract Transform Load</td>
</tr>
<tr>
<td>RA</td>
<td>Relational Algebra</td>
</tr>
<tr>
<td>LA</td>
<td>Linear Algebra</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time</td>
</tr>
<tr>
<td>PCA</td>
<td>Principal Component Analysis</td>
</tr>
<tr>
<td>ADT</td>
<td>Algebraic Data Types</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
</tbody>
</table>
1 Introduction

Requirements of data analytics have changed over the last decade. Traditionally confined to aggregation queries over relational data, modern analytics is focused on advanced in-situ analysis of dirty and unstructured data at scale. Data sources such as log files, clickstreams, etc., are first cleansed using relational operators, and then mined using clustering and classification tasks based on statistical and machine learning (ML) methods. As a result, data cleaning and pre-processing stand typically as initial step of advanced data analysis pipelines (e.g., product recommendations, statistical analysis). Moreover, the pre-processing logic and data representation often depend on the ML method that will be applied subsequently.

While relational domain-specific languages (DSLs) such as Pig [14], Hive [15], or Spark SQL/DataFrame [3] are a good fit for ETL tasks, programming machine learning algorithms in those languages is cumbersome. To this end, R-like DSLs such as SystemML’s DML [10] or Apache Mahout’s Samsara were proposed. These DSLs offer linear algebra and control flow primitives suitable for expressing ML algorithms, but only provide limited, non-intuitive support for classic ETL tasks. This strict separation of programming paradigms introduces three fundamental problems.

- **Impedance Mismatch.** Large analysis pipelines have to be authored in multiple DSLs, plumbed together by additional code, and possibly executed on different systems.
- **Inefficient Data Exchange.** Pre-processing in a relational algebra (RA) DSL and subsequent learning in a linear algebra (LA) DSL enforces materialization of the intermediate results at the boundary. Moreover, unless the staging format offers meta-information about the physical layout, this information is lost when the boundary is crossed.
- **Loss of Optimization Potential.** Separating RA and LA into distinct DSLs entails that different, albeit similar, intermediate representations (IRs) and compilation pipelines are used for the two languages. For example, compare the compilation phases and DAG-based IRs for Hive [15] and SystemML [4]. As a result, optimizations that could be applied among the IRs (e.g., filter and projection push-down, sharing of physical layout) are currently not possible.

To overcome these problems, we argue for the unification of relational and linear algebra into a common theoretical foundation. To achieve this goal, first we need to explore and reason about optimizations across the two algebras in a suitable intermediate language representation (IR). Second, we need to showcase the added benefits of unification and the optimizations that come thereof, by defining a common DSL with high-level programming abstractions for both relational and linear algebra. In line with the benefits offered by other UDF-heavy dataflow API’s, the proposed DSL should be embedded in a host-language like Scala (e.g. Spark RDDs, Samsara) rather than external ones (e.g., Pig, DML).

In this deliverable, we show a first prototype of Lara, an embedded DSL in Scala that offers abstract data types for both relational and linear algebra (i.e., bags and matrices). We build our work on Emma [1, 2], a DSL for scalable collection based processing, which we extend with linear algebra data types. We exploit the code rewriting facilities of the Scala programming language to lift a user program into a unified intermediate representation for joint optimization of linear and relational algebra. Given this IR, a just-in-time (JIT) compiler generates code for different execution engines (Spark and Flink).

The code of Lara first stable prototype is available at the following GitHub repository: https://github.com/proteus-h2020/proteus-language. The new version of Lara will be pushed to the same repository in the future.
1.1 Motivating example

Consider a set of machinery sensors found in industrial plants taking part in the production of home mixers. In the end of the production line, a percentage of those mixers is found to be defective. The goal of our analysis is to train a classifier which will predict whether a mixer has high chances of being defective, based on the given measurements. For this task, we have to gather data from various log files residing in different production plants and join them in order to get all measurements of a mixer throughout its production. Since there are thousands of measurements per mixer and millions of mixers, we first run a Principal Component Analysis (PCA) for feature reduction. The above process can be implemented in Lara as shown in Listing 1. Lines 2 and 3 read the data from two different industrial plants before joining them (lines 7-11) to gain a full view over all measurements for each of the mixers. Note that the join is expressed as a native Scala construct, called for-comprehension (see [2] for details). Next, DataBag X which contains all projected measurements \((a_1, ..., a_N, b_1, ..., b_M)\) is converted into Matrix \(M\) (line 14). The next line computes the mean of each of the measurement columns before we compute matrix \(U\), holding the deviation of each of \(M\)'s cells from their corresponding column’s mean using the fill operator (called “filling function” in [13]). Finally, line 22 computes the covariance matrix \(C\). In the next step, we would feed matrix \(C\) to the PCA algorithm, which is omitted from the example.

```scala
// read measurements into the DataBags A and B
val A = readCSV(...) //
val B = readCSV(...) //
// SELECT a_1, ..., a_N, b_1, ..., b_M
// FROM A, B
// WHERE A.id = B.id
val X = for {
    a <- A
    b <- B
    if a.id == b.id
    } yield (a_1, ..., a_N, b_1, ..., b_M)

// Convert the DataBag X into a Matrix
val M = X.toMatrix()
// Calculate the mean of each column of the matrix
val means = for (c <- M.cols()) yield mean(c)
// Compute the deviation of each cell of M
// to the cell's column mean.
val U = M - Matrix.fill(M.numRows, M.numCols)
    ((i,j) => means(j))
// compute covariance matrix
val C = 1 / (U.numRows - 1) * U.t %*% U
// compute singular value decomposition
// e.g. rescale M, reduce dimensions, etc.
```

Listing 1. Code snippet written in Lara

1.2 Discussion

Observe that the ETL part of the pipeline is expressed as a declarative program of transformations over
DataBags, whereas the ML part is expressed in linear algebra. Moreover, no physical execution strategy has been pre-determined by the programmer. Our matrix abstraction is strongly influenced by R’s matrix package and includes all common operations on matrices. In addition, there are two explicit operations to convert from a DataBag to a Matrix and vice versa. Finally, note that converting a DataBag into a Matrix does not necessarily mean that an operation is going to take place on a physical Matrix representation. For example, consider a scalar multiplication of a Matrix: the multiplication could be applied directly on a DataBag, since scalar multiplications do not rely on special, linear algebra-specific operators. Thus, we let the optimizer decide to apply which physical data representation on which operations.

1.3 Document structure

After a brief introduction in Section 2, related works are investigated. We divide it into two subcategories being ML Libraries and Languages and Algebra Unifying Approaches. The foundations of declarative language are presented in Section 3, where core concepts like User Types, System Types, Type Conversions, and Control-flow are explained. Section 4 presents optimizations in the language. Implementation details are described in section 5. Finally, Lara API is shown in Section 5 along with some working example.
2 Related work

- **ML Libraries & Languages.** SystemML’s DML [4] and Mahout’s Samsara provide R-like linear algebra abstractions and execute locally or distributed on Hadoop and Spark. While Samsara has fixed implementations for its linear algebra operations, SystemML applies inter-operator optimizations like operator fusion and decides execution strategies based on cost estimates. As there is no support for relational operators, ETL has to be executed in a different system and there is no potential for holistic optimization. The Delite project [6] provides a compiler framework for domain-specific languages targeting parallel execution. Delite’s IR is based on higher-order parallelizable functions (e.g., reduce, map, zipWith). However, Delite’s IR does not allow to reason holistically about linear and relational algebra. In this work, we base our reasoning on types and on the holistic view of the AST. Finally, we believe that monad comprehensions provide a better formal ground for reasoning and applying algebraic optimizations.

- **Algebra Unifying Approaches.** Kumar et al. [12] introduce learning over linear models on data residing in a relational database. They push parts of the computation of the ML model into joins over normalized data, similar to [7]. These works focus on generalized linear models, as we focus on more generic optimizations that can be derived directly from the common intermediate representation of linear and relational algebras. MLBase [11] provides high-level abstractions for ML tasks with basic support for relational operators. Their DSL enables the optimizer to choose different ML algorithm implementations, but does not take relational operators into account nor does it apply any optimization among algebras.
3 Language Foundations

Types as First Class Citizens. Our DSL is based around the concept of generic types. We propose a set of elementary generic types that model different structural aspects and represent both user-facing (e.g., matrix, bag) and engine-facing (e.g., partitioning) abstractions. Each type implies: (i) a set of structural constraints expressed as axioms, (ii) a set of structure-preserving transformations that operate element-wise and correspond to the functional notion of map. Moreover, the types can be suitably composed to define new, richer structure. This allows reasoning about the optimization space in a systematic way. The following concepts lie in the core of the language foundations.

- **User Types.** The core types included in our model are Bag A, Matrix A, and Vector A. These types are polymorphic (generic) with a type argument A and represent different containers for data (i.e., elements) of type A. As such, their implementations should be independent of A. Categorically, generic types correspond to the notion of free functors. An important class of free functors which are of particular interest for us are the so-called polynomial functors. Those can be used to define generic data types (called algebraic data types (ADTs)) by means of enumerating primitive functions that construct values of those types. Emma [3], for example, conceptually treats bags as an ADT using the so-called union representation.

\[ \text{data Bag} A = \text{empty} \mid \text{sng} A \mid \text{union} \text{Bag} A \times \text{Bag} A \]

Bag values can be constructed by a nullary constructor (empty), an unary constructor (sng), or a binary constructor (union), and the associated axioms state that empty is a unit and union is associative and commutative. LA types with fixed dimensions, such as Vector\(^n\)A and Matrix\(^n\times m\)A, as well as the monoids used in [9] naturally fit this framework. The signature and dependencies between the constructors in an ADT definition impose certain type structure. Mappings that preserve this structure are called homomorphisms. In the case of generic types, the structure represents an abstract model for the shape of the container, and homomorphisms are characterized by a second-order function called map. It is important to note that map is predominantly associated with collections nowadays (as in MapReduce), while the concept of map is pervasive to all generic types.

- **System Types.** Reasoning about optimizations that affect physical aspects such as partitioning or blocking means that those should be included in our model. Crucially, this type of structure can also be represented by generic types. For example, we can use “Par T A” to represent a partitioned container of type T A (e.g., Bag A, Matrix A), and Block\(^n\)A to represent square-shaped blocks with dimension n. Homomorphisms (maps) over those, model partition- or block-preserving function applications (e.g., corresponding to mapValues in Spark’s RDD API) respectively.

- **Type Conversions.** An obvious candidate to formalize generic type conversions in a categorical setting are natural transformations – polymorphic functions \( t : T A \to U A \) which change the container type from T to U (e.g., from Matrix to Bag) irrespective of the element type A. Their characteristic property states that the application of \( t \) commutes with the application of map \( f \) for all \( f \). This formalism, however, cannot be directly applied in all cases. For example, toMatrix : Bag \((N, A)\) \(\to\) Matrix A preserves the element value A but relies on an extra index N to determine the element position in the resulting matrix. Extending or adapting the theory of natural transformations in order to fit our needs is an open research question.
• **Control-Flow.** To enable rewrite-based optimizations, our proposed language is referentially transparent (i.e., purely functional). Moreover, in order to facilitate efficient and concise implementation of optimizations, the language IR should satisfy the following requirements: (R1) Both elementary and compound expressions should be addressable by name; (R2) Each *use-def* and *def-use* information should be efficient, and easy to compute and maintain; and (R3) Control and data flow should be handled in a unified way.

All the above requirements could be satisfied by an IR in *static single assignment* (SSA) form. Graph-based SSA IRs (e.g., sea of nodes) are nowadays used in compiler backends like LLVM and Java HotSpot. We plan to use a purely functional IR that conforms to the SSA constraints. It enforces R1 through a restriction on the allowed terms called *A-normal form*, and R2-R3 by modelling control-flow through function calls in the so-called *direct style*. 
4 Optimizations

Several holistic optimizations can be derived from the unified formal model and implemented under the assumption of a full view of the algorithm code. One example is projection push down. To enable this optimization, we analyse the code to identify those data fields that are never accessed, and thus, are not needed during data processing. Another example deals with filters, i.e., filtering operations performed on the matrix object are conversely executed on the original DataBag. In the sequel, we present more sophisticated optimizations that come from a deeper analysis of a program’s code.

4.1 Matrix Blocking Through Joins

Distributed operations over matrices are commonly done over block-partitioned matrices [10, 5]. This representation differs a lot from the unordered, non-indexed bag representation, commonly used in dataflow APIs.

Consider again the example in Listing 1. Lines 7-11 perform a join, producing a bag that is converted to a matrix and processed in lines 16 and 22. Note that the subsequent linear algebra operations (filling as well as computing the covariance matrix) can be executed over a block-partitioned matrix. A naive execution of this program would require to: shuffle the data once to execute the join, and then shuffle another time to block-partition the matrix in order to perform the linear algebra operations. In the sequel, we use an example to show (i) how the naive approach would perform the join and subsequently the block partitioning, and (ii) how we can avoid one partitioning step (for the join).

Naive Approach. Assume we execute the join of A and B as shown in Figure 1d on 4 nodes, using hash-partitioning with \( h(k) = k \mod 4 \), where \( k \) is the product id. To block partition, the matrix for the subsequent linear algebra operations, systems typically introduce a surrogate key \( \text{rowIdx} \) for each tuple of the join result, to assign subsets of the rows and columns to blocks. Therefore, we assign the following key to each tuple:

\[
k = \left( \frac{\text{rowIdx}}{\text{rowBlkSize}}, \frac{\text{colIdx}}{\text{colBlkSize}} \right)
\]

In the above formula, \( \text{colIdx} \) represents the column index of the tuple, \( \text{rowBlkSize} \) is the number of rows in the block, whilst, \( \text{colBlkSize} \) is the number of columns in the same block. The result of this key assignment for the join result is depicted in Figure 1b. Note that the blocks are partial. A grouping operation can bring the partial blocks on the respective machines and construct the final blocks as shown in Figure 1f.

Partition Sharing. A full view over the code in Listing 1 allows us to see both, the RA part in lines 1-11, the LA part in lines 16-22, and to holistically reason about the type conversion in line 14. Ideally, the join operation and the linear algebra operations can share the same partitioning. We can achieve that by range-partitioning the input tables \( A \) and \( B \) separately and then combine them. More specifically, we use a different key to partition the inputs, considering both the (unique, and sequential\(^1\)) product id and the column index:

\(^1\) Similar optimizations apply on joins over non-unique keys (e.g. normalized data [13]). Moreover, the assumption of sequential primary keys can be relaxed in the expense of an extra pass over the data that is negligible in complex analysis programs.
\[ k = \left( \frac{\text{ID}}{\text{rowBlkSize}}, \frac{\text{resultColIdx}}{\text{colBlkSize}} \right) \]

where \( \text{resultColIdx} \) is the index of the column in the (now virtual) bag \( X \) and \( \text{ID} \) is the primary key of the inputs. As the schema of the join result is explicitly provided in \( \text{Lara} \) (Listing 1 line 11), we can easily obtain the column indexes of the join result (X). The partitioning of the tables is shown in Figure 1e. Observe that the blocks with column index 0 are split across the tables, thus, we also have column-wise partial blocks. To create the final partitioning with complete blocks, we union \( A \) and \( B \), before we aggregate the blocks sharing the same block id.

4.2 Row-wise Aggregation Pushdown

In our example Listing 1, line 16 calculates the mean for each column in the matrix. Now, let’s consider calculating the mean for each row, as shown in the following snippet:

```scala
// Convert the DataBag X into a Matrix
val M = X.toMatrix()

// Calculate the mean of each row of the matrix
val means = for (r <- M.rows()) yield (mean(r))
```

This would require a full pass over the data, and in fact, as the matrix is partitioned block-wise, we have to merge the partial aggregates of the blocks, to compute the full aggregate over each row. On the other hand, this could be executed in the \( \text{DataBag} \) representation in a single pass, as we have tuple/row at-a-time processing. In such a scenario, the aggregate could be computed while executing the join with...
a simple mapper after the join.

### 4.3 Caching Through Iterations

A holistic view over the program allows us to reason about the structure of the control flow. For instance, caching data, which is used repeatedly within an iteration or along multiple control-flow branches, can result in great performance benefits. This becomes even more interesting when the data originates from pre-processing which would otherwise be re-computed for each iteration. The decision to cache is not always evident and forces the programmer to consider system specifics – *Lara* currently employs a greedy strategy that implicitly caches data used repeatedly in iterations.
5 Implementation Details

Since Lara is based on Emma DSL [1], we structure this section as follows: we first explain the technical details of Emma, and then we describe how we implemented Lara.

5.1 Emma: a Scala DSL for scalable data analysis

The following section is based on Emma description on its official github web-page\(^2\).

Emma’s main goal is to improve developer productivity by hiding parallelism aspects behind a high-level, declarative API and through deep reuse of native Scala syntax and constructs. Emma supports state-of-the-art dataflow engines such as Apache Flink and Apache Spark as backend co-processors.

Programs written for distributed execution engines usually suffer from some well-known pitfalls:

1. Details and subtleties of the target engine **execution model** must be well understood to write efficient programs.
2. Due to the number of abstraction leaks, program code can be **hard to read**.
3. **Opportunities for optimization** are missed out due to hard-coded execution strategies or isolated (per-dataflow) compilation.

*Emma offers* a declarative API for parallel collection processing. *Emma* programs benefit from **deep linguistic re-use** of native Scala features such as for-comprehension, case-classes, and pattern matching. Programs written in *Emma* are **analysed and optimized holistically** for data-parallel execution on a co-processor engine such as Flink or Spark.

Emma’s parallel execution is based on DataBag[A] core type, which abstracts a distributed collection of objects of type A. This collection may be unordered and contain duplicates.

DataBag[A] abstracts a processing model called **structural recursion**. DataBag[A] instances are built by composition of Empty bags, Singleton Bags, and unioned bags. This procedure is specified as the fold second order function as follows:

```scala
def fold[B](e: B)(s: A => B, u: (B, B) => B): B = this match {
  case Empty          => e
  case Singleton(x)   => s(x)
  case Union(xs, ys)  => u(xs.fold(e)(s, u), ys.fold(e)(s, u))
}
```

where e is the empty bag, s produces the singleton bag and u joins two bags.

Many collection processing primitives can be derived from a fold operation, e.g:

```scala
val size = xs.fold(0L)(const(1L), _ + _)
val min  = xs.fold(None)(Some(_), lift(_ min _))
val sum  = xs.fold(0)(identity, _ + _)
```

Emma offers pre-defined aliases for such common folding operation:

<table>
<thead>
<tr>
<th>Fold Alias</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce, reduceOption</td>
<td>General</td>
</tr>
<tr>
<td>isEmpty, nonEmpty, size</td>
<td>Cardinality</td>
</tr>
</tbody>
</table>

\(^2\)https://github.com/emmalanguage/emma
Joins and cross products in *Emma* can be declared using for-comprehension syntax in a manner akin to *Select-From-Where* expressions known from SQL.

```scala
for {
  email   <- emails
  sender  <- people
  receiver <- people
  if email.from == sender.email
  if email.to  == receiver.email
} yield (email, sender, receiver)
// DataBag[(Email, Person, Person)]
```

In addition to the collection manipulation primitives presented above, *Emma* offers a `groupBy` combinator, which (conceptually) groups bag elements by key and introduces a level of nesting.

To compute the average size of email messages by sender, for example, one can write the following straightforward expression.

```scala
for {
  Group(sender, mails) <- emails.groupBy(_.from)
} yield {
  val sum  = mails.map(_.msg.length).sum
  val cnt  = mails.size
  sum / cnt
}
```

If the above computation were expressed in a similar way in Flink’s or Spark’s APIs it would result in dataflow graphs that do not scale well with the size of the input. Emma takes a holistic optimization approach which allows to transparently rewrite the program into scalable Flink/Spark dataflows.

The presented API is not abstract. The semantics of each operator are given directly by the default standalone implementation. This allows the end-user to incrementally develop, test, and debug data analysis algorithms at small scale locally as a pure Scala program.

For example, the following code-snippet that counts words runs out-of-the-box as a normal Scala program.

```scala
val words = for {
  line <- DataBag.readText(input)
  word <- DataBag(line.toLowerCase.split("\W+"))
} yield word
// group the words by their identity
// and count the occurrences of each word
val counts = for {
  Group(word, occs) <- words.groupBy(identity)
} yield word -> occs.size
// write the results into a CSV file
counts.writeCSV(output, CSV())
```
The above code can be wrapped in one of the following two Scala macro and then be executed on either Spark or Flink.

```scala
emma.onSpark {
  // word count code from above goes here!
}
emma.onFlink {
  // word count code from above goes here!
}
```

The backend macros identify all DataBag[A] terms in the quoted code fragment and re-write them jointly to maximize the degree of parallelism. For example, the groupBy and the subsequent folds from the nesting example are executed using more efficient target-runtime primitives like reduceByKey. The holistic translation approach also allows us to transparently insert co-processor primitives like cache, broadcast, and partitionBy based on static analysis of the quoted code.

### 5.1.1 Compiler Infrastructure

Emma compiler is based on transformation pipelines that operate on Scala Abstract Syntax Tree. A bundle of transformations is provided in Emma compiler, which can be chained together to build a new pipeline. The purpose of those transformations is to lift code written in Emma Source Language to Emma Core Language. The lift transformation is quotation-based, i.e., a Scala macro accesses the Scala Abstract Syntax Tree (AST) of the quoted code and then the compiler performs several transformations on this AST to obtain a suitable intermediate representation (IR) called Emma Core Language. Emma IR is further optimized and then translated to the desired runtime, e.g., Apache Spark, Apache Flink.

The rewrite on the Emma Core Language transparently inserts partial aggregates whenever possible and thereby re-moves expensive group materializations. 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 that fuses the group construction performed by the groupBy together with the subsequent fold applications on the
group values. For those rewrites, there are two types of optimization, i.e., banana split and fold-build fusion. 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. The fold-build fusion 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.

Built-in Emma transformations are classified as follows:

<table>
<thead>
<tr>
<th>Language Transformation</th>
<th>Transformation Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source → Source_{ANF}</td>
<td>Administrative Normal Form</td>
<td>To destruct composite terms such that arguments are simple literals or identifiers.</td>
</tr>
<tr>
<td>Source → LNF</td>
<td>Direct-Style control flow</td>
<td>To eliminate while and do-while loops, if-else branches, and local variables, and replaces them with nested, mutually recursive local methods and calls.</td>
</tr>
<tr>
<td>Source → LNF</td>
<td>Let-Normal Form</td>
<td>Composition of Administrative Normal Form and Direct-Style.</td>
</tr>
<tr>
<td>LNF → Core</td>
<td>Resugar</td>
<td>Add comprehension syntax as a first-class citizen in by <em>re-sugaring</em> monad operators for a monad type M.</td>
</tr>
<tr>
<td>Core → LNF</td>
<td>Desugar</td>
<td>Opposite of Resugar.</td>
</tr>
<tr>
<td>LNF → LNF’</td>
<td>Reference (Un)Inlining</td>
<td>Expand / reduce the type of terms that can appear in the expr position of let blocks by inlining references.</td>
</tr>
<tr>
<td>Core → Core’</td>
<td>Normalize</td>
<td>Normalizes comprehensions for a monad type M by iteratively unnesting nested comprehensions appearing in the right term of a generator or in the head position.</td>
</tr>
<tr>
<td>Source → Core</td>
<td>Lift</td>
<td>The canonical way to lift Emma expressions from the Source to Core language.</td>
</tr>
</tbody>
</table>
The Lift operation is defined as the composition of the others and its behaviour can be summarized as follows:

### 5.1.2 Emma Source Language

The source language of Emma is a subset of Scala with a couple of restrictions, which are needed:

- to ensure that the quoted source code can be translated and efficiently executed on dataflow engines:
  - object mutation to prevent race conditions
  - closure modification to ensure serializability of lambda functions
  - class and object definitions that can be externally defined
  - local method definition to disallow general recursion
- to disable constructs that require more involved translation, i.e., non-local control-flow:
  - explicit return statements
  - throwing and handling of exceptions
  - refutable pattern matching
- to semantically support laziness and side effects due to A-Normal Form of Emma Core Language:
  - call-by-name parameters are evaluated eagerly and only once, because intermediate terms are assigned to vals. Therefore, the source program should not contain side effects in call-by-name code.
  - side effects in general might be eliminated as dead code or reordered.
5.2 Lara Internals

Lara is built on top of Emma. Lara extends Emma’s DataBag operations by providing a new method that allows to transform a DataBag to a Matrix and vice versa.

Lara API closely resembles languages such as R, Matlab and libraries such as NumPy (Python). This was done intentionally: data science experts are accustomed to such declarative languages and in PROTEUS we want to make sure that non-systems experts can develop scalable machine learning algorithms.

The Matrix API includes Matrix-Scalar (e.g., addition of scalar in Matrix), Matrix-Vector (e.g., Matrix-Vector multiplication) and Matrix-Matrix operations (e.g., Matrix-Matrix multiplication). Finally, the API includes transformations of Matrices (e.g., matrix transpose), filling operations (e.g, diagonal matrix) and finally extraction operations (e.g., selecting out columns or rows).

5.2.1 Matrix-Scalar Pointwise Operators

```python
def +(that: A): Matrix[A]
def -(that: A): Matrix[A]
def *(that: A): Matrix[A]
def /(that: A): Matrix[A]
```

5.2.2 Matrix-Vector Pointwise Operators

```python
def +(that: Vector[A]): Matrix[A]
def -(that: Vector[A]): Matrix[A]
def *(that: Vector[A]): Matrix[A]
def /(that: Vector[A]): Matrix[A]
```

5.2.3 Matrix-Matrix Pointwise Operators

```python
def +(that: Matrix[A]): Matrix[A]
def -(that: Matrix[A]): Matrix[A]
def *(that: Matrix[A]): Matrix[A]
def /(that: Matrix[A]): Matrix[A]
```

5.2.4 Binary Matrix-Matrix and Matrix-Vector Operators

```python
def %%(that: Matrix[A]): Matrix[A]
```
def %%%(that: Vector[A]): Vector[A]

5.2.5 Matrix self-transformations Operators

def diag(): Vector[A]
def transpose(): Matrix[A]
def row(rowIndex: Int): Vector[A]
def column(colIndex: Int): Vector[A]

def rows[B: Numeric : ClassTag](f: Vector[A] => Vector[B]): Matrix[B]
def indexedRows[B: Numeric : ClassTag](f: Idx[Int, Vector[A]] => B): Vector[B]
def indexedRows[B: Numeric : ClassTag](f: Idx[Int, Vector[A]] => Vector[B]): Matrix[B]
def cols[B: Numeric : ClassTag](f: Vector[A] => Vector[B]): Matrix[B]
def indexedCols[B: Numeric : ClassTag](f: Idx[Int, Vector[A]] => B): Vector[B]
def indexedCols[B: Numeric : ClassTag](f: Idx[Int, Vector[A]] => Vector[B]): Matrix[B]
def elements[B](z: B)(s: A => B, p: (B, B) => B): B

private[emma] def map[B: Numeric : ClassTag](f: A => B): Matrix[B]

private[emma] def plus(other: Matrix[A]): Matrix[A]
5.3 Example Algorithms implemented in Lara

In this Section, we show how two more complex data analytic algorithms, namely, K-means clustering and logistic regression can be implemented in Lara.

5.3.1 K-Means

```scala
import eu.proteus.lara.{Matrix, Vector} // load Lara API
/**
 * @param x matrix representing the points. Each row corresponds to a point
 * @param k number of clusters
 * @param maxIterations maximum steps before the algorithm stops if it has not converged
 * @param epsilon convergence criterion
 * @return centroids matrix with k rows where each row corresponds to a trained centroid
 */
def kMeans(x: Matrix[Double], k: Int, maxIterations: Int, epsilon: Double): Matrix[Double] = {
  val m = x.numRows
  val n = x.numCols
  var change = 0.0
  var currentIteration = 0

  //initialisation of the centroids with k random points
  var centroids : Matrix[Double] = Matrix.fillColumns(n,k)(j => x.row(Random.nextInt(m)))
  var newCentroids : Matrix[Double] = centroids

  do {
    //computation of the matrix of the square distances
    val distances : Matrix[Double] = Matrix.fill(m,k)((i,j) => (x.row(i) - newCentroids.column(j)).fold(0.0)(s => s * s, (l,r) => l + r))

    //get the id of the nearest centroid for each point
    val minIdx = Vector.fill(m)( i =>
      distances.row(i).indexedFold((-1,Double.MaxValue))(s => (s.id,s.value),
        (l,r) => if (l._2 < r._2) l else r)._1
    )

    //computation of the matrix indicating where the points belong
    val z : Matrix[Double] = Matrix.fill(m,k)(
      (i,j) => if (j == minIdx.get(i)) 1.0 else 0.0
    )

    //vector indicating the size of each cluster
    val clusterSizes = Vector.fill(k)(i => z.column(i).aggregate(_ + _))

    //compute new centroids
    newCentroids = newCentroids
    newCentroids = (x.transpose() %*% z).indexedCols(idx =>
      if (clusterSizes.get(idx.id) == 0.0) x.row(Random.nextInt(m))
      else idx.value / clusterSizes.get(idx.id)
    )
    currentIteration += 1

    //compute change
    change = Vector.fill(k)(
      i => (centroids.column(i) - newCentroids.column(i)).fold(0.0)(
        s => s * s, (l,r) => l + r
      ).aggregate(_ + _)
    )
  } while (change > epsilon && currentIteration < maxIterations )
```
newCentroids.transpose() // return the trained model
}

### Listing 2 K-Means implementation in Lara

#### 5.3.2 K-Means example

```scala
val data = ...  
val maxIterations = 100  
val epsilon = 0.000001  

// compute the centroids for K-Means (k=5)  
val centroids = KMeansClustering.kMeans(data, 6, maxIterations, epsilon)
```

### Listing 3 K-Means example

#### 5.3.3 Linear Regression (with Gradient Descent optimization method)

```scala
import eu.proteus.lara // load Lara API

/**  
 * @param X matrix representing the points. Each row corresponds to a point  
 * @param y target variables  
 * @param betas initial weights  
 * @param alpha learning rate  
 * @param numIterations number of training epochs  
 * @return weights of the fitted model  
 */

def gradientDescent(X: Matrix, y: Vector, betas: Vector,  
                     alpha: Double, numIterations: Int): Vector = {

  val m = X.numRows  
  var newBetas = betas  

  for (i <- 0 until numIterations) {
    val hypothesis = X %*% newBetas // compute current hypothesis/activations  
    val loss = hypothesis - y // compute loss  
    val cost = loss.map(x => x * x).aggregate((a, b) => a + b) / (2 * m) // compute avg cost per example  
    val gradient = (X.transpose() %*% loss) / m // compute new gradient  

    // update parameters  
    newBetas = newBetas - (gradient * alpha)
  }

  // return the trained model  
  newBetas
}
```

### Listing 4 Linear Regression with Gradient Descent optimization method

#### 5.3.4 Linear Regression Example

```scala
val (x, y) = ...  
val m = x.numRows  
val n = x.numCols  
val numIterations = 100000  
val alpha = 0.0005  
val betas = Vector.ones[Double](n)

val result = LinearRegressionGD.gradientDescent(x, y, betas, alpha, numIterations)
```
5.4 Follow-up: online machine learning abstractions

Lara API may be embedded in some abstraction for Online Machine Learning. The purpose of this abstraction is to let end-users focus on *what the algorithm executes* rather than *how to execute the algorithm*. Such abstractions can be further embedded in an Emma quotation in order to be fully integrated with PROTEUS ecosystem. These abstractions will be fully described in a future Deliverable (D4.5).
6 Conclusion

In this document, we present a first prototype of a declarative language, Lara, with final syntax definition, language foundations, optimizations used and sample APIs.

Lara is an embedded DSL in Scala, which offers abstract data types for both relational and linear algebra (i.e., bags and matrices). We build our work on Emma [1, 2], a DSL for scalable collection based processing, which we extend with linear algebra data types. We exploit the code rewriting facilities of the Scala programming language to lift a user program into a unified intermediate representation for joint optimization of linear and relational algebra. Given this IR, a just-in-time (JIT) compiler generates code for different execution engines (Spark and Flink). An extensive description of Emma’s capabilities is also provided.

One important concept in Lara’s language foundations is User Types. The core types included in our model are Bag A, Matrix A, and Vector A. These types are polymorphic (generic) with a type argument A and represent different containers for data (i.e., elements) of type A. Reasoning about optimizations that affect physical aspects such as partitioning or blocking means that those should be included in our model, which are System Types in language definition. Another concept in language foundations of Lara is Type Conversions that are natural transformations – polymorphic functions and Control-Flows.

A number of holistic optimizations can be derived from the unified formal model and implemented under the assumption of a full view of the algorithm code, e.g., projection push down and filters. Projection push down removes fields that would be never accessed during the program execution. Filtering operations applied on matrix are more efficiently executed on DataBags. In the sequel, we present more sophisticated optimizations that come from a deeper analysis of a program’s code provided in document.

Finally, a first prototype of Lara APIs and example algorithms are provided to get overall intuition of Lara. The API closely resembles languages such as R, Matlab and libraries such as NumPy (Python). This was done intentionally: data science experts are accustomed to such declarative languages and in PROTEUS we want to make sure that non-systems experts can efficiently develop scalable machine learning analytics.
References


