D3.6 Syntax definition for a declarative language based on machine learning requirements

Authors: Asterios Katsifodimos, Jeyhun Karimov, Bonaventura del Monte

<table>
<thead>
<tr>
<th>Deliverable nature:</th>
<th>Report (R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dissemination level: (Confidentiality)</td>
<td>Public (PU)</td>
</tr>
<tr>
<td>Contractual delivery date:</td>
<td>30.11.2016</td>
</tr>
<tr>
<td>Actual delivery date:</td>
<td>25.11.2016</td>
</tr>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
<tr>
<td>Total number of pages:</td>
<td>24</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 in order 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 deliverable 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 design of Lara, a deeply embedded language in Scala which 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) which enables optimizations across linear and relational algebra. The IR is finally used to compile code for different execution engines.
Executive summary

In this deliverable, we define 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 is overlapping with PROTEUS project’s one of main missions, to investigate and develop ready-to-use scalable online machine learning algorithms. There is abundant amount of machinery sensors ArcelorMittal. Based on data received from these sensors the system should

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

This can be expressed in *Lara*, expressing ETL part of the pipeline as a declarative program of transformations over *DataBags*, whereas the ML part in linear algebra without any need to determine physical execution strategy by the programmer.

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, as well as *(ii)* a set of structure-preserving transformations, which operate element-wise and correspond to the functional notion of map.

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. Examples are projection push down (based on knowledge that fields are never accessed), as well as filters that are e.g., applied on matrices whereas they can be pushed to the *DataBags* from which the matrices originate. In the sequel we present more sophisticated optimizations that come from a deeper analysis of a program’s code.
Document Information

<table>
<thead>
<tr>
<th>IST Project Number</th>
<th>Acronym</th>
<th>PROTEUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>687691</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Full Title | | | |
|------------|-----------------------------|
| Scalable online machine learning for predictive analytics and real-time interactive visualization | | |

| Project URL | | | |
|-------------|-----------------------------|
| http://www.proteus-bigdata.com/ | | |

| EU Project Officer | | | |
|--------------------|-----------------------------|
| Martina EYDNER | | |

| Deliverable Number | Title | | |
|--------------------|-------|-----------------------------|
| D3.6               | Syntax definition for a declarative language based on machine learning requirements | | |

| Work Package Number | Title | | |
|--------------------|-------|-----------------------------|
| WP3                | Scalable Architectures for both data-at-rest and data-in-motion | | |

| Date of Delivery | Contractual | Actual | | |
|------------------|-------------|--------|-----------------------------|
| M12              | M12         |        | |

| Status | | | |
|--------|-----------------------------|
| version 1.0 | final □ | |

| Nature | | | |
|--------|-----------------------------|
| report | | |

| Dissemination level | | | |
|--------------------|-----------------------------|
| public | | |

| Authors (Partner) | | | |
|-------------------|-----------------------------|
| Name | Asterios Katsifodimos | E-mail | asterios.katsifodimos@tu-berlin.de |
| Partner | DFKI | Phone | 0171 549 5731 |

Abstract (for dissemination)

Advanced data analysis typically requires some form of pre-processing in order 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 design of Lara, a deeply embedded language in Scala which 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) which 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.

**Version Log**

<table>
<thead>
<tr>
<th>Issue Date</th>
<th>Rev. No.</th>
<th>Author</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.11.2016</td>
<td>1.0</td>
<td>Hamid Bouchachia</td>
<td>Minor changes</td>
</tr>
</tbody>
</table>
# Table of Contents

Executive summary .................................................. 4  
Document Information ............................................. 5  
Table of Contents .................................................... 7  
List of figures and/or list of tables ............................... 8  
Abbreviations ........................................................... 9  

1 Introduction ......................................................... 10  
   1.1 Motivating example ......................................... 11  
   1.2 Discussion ....................................................... 11  
   1.3 Document structure ........................................... 12  

2 Related work ....................................................... 13  

3 Language Foundations .......................................... 14  

4 Optimizations ...................................................... 16  
   4.1 Matrix Blocking Through Joins ............................ 16  
   4.2 Row-wise Aggregation Pushdown ........................ 17  
   4.3 Caching Through Iterations ................................. 18  

5 Concrete API ....................................................... 19  
   5.1 Matrix-Scalar Pointwise Operators ....................... 19  
   5.2 Matrix-Vector Pointwise Operators ....................... 19  
   5.3 Matrix-Matrix Pointwise Operators ....................... 19  
   5.4 Binary Matrix-Matrix and Matrix-Vector Operators . 19  
   5.5 Matrix self-transformations Operators .................. 20  

6 Example Algorithms implemented in Lara .................... 21  
   6.1 K-Means .......................................................... 21  
   6.2 Linear Regression .............................................. 22  

7 Conclusion ........................................................... 23  

References ............................................................ 24
List of figures and/or list of tables

Figure 1. Two execution strategies for $\text{toMatrix}(A \bowtie B)$ (colors represent different partitions). Above (a-c) Naive Approach. Below (d-f) Partition sharing................................. 17

Listing 1. Code snippet written in Lara................................................................. 11
### 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

Data analytics requirements 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 is typically an initial step of advanced data analysis pipelines (e.g., product recommendations, statistical analysis). Moreover, the pre-processing logic and data representation often depends 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 of 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 (e.g., Pig, DML).

In this deliverable we propose Lara, 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).
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.

Listing 1. Code snippet written in Lara

```
1 // read measurements into the DataBags A and B
2 val A = readCSV(...) //
3 val B = readCSV(...) //
4 // SELECT a_1, ..., a_N, b_1, ..., b_M
5 // FROM A, B
6 // WHERE A.id = B.id
7 val X = for {
8   a <- A
9   b <- B
10  if a.id == b.id
11  } yield (a_1, ..., a_N, b_1, ..., b_M)
12
13 // Convert the DataBag X into a Matrix
14 val M = X.toMatrix()
15 // Calculate the mean of each column of the matrix
16 val means = for ( c <- M.cols() ) yield (mean(c))
17 // Compute the deviation of each cell of M
18 // to the cell's column mean.
19 val U = M - Matrix.fill(M.numRows, M.numCols)
20   ((i,j) => means(j))
21 // compute covariance matrix
22 val C = 1 / (M.numRows - 1) * U.t %*% U
23 // compute singular value decomposition
24 // e.g. rescale M, reduce dimensions, etc.
```

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 in which physical data representation operations apply.

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 is explained. Section 4 presents optimizations in the language. Finally, concrete API is shown in Section 5.
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 allows 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, as well as (ii) a set of structure-preserving transformations, which 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 for reasoning about the optimization space in a systematic way. The following concepts lie in the core of 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 on 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, advocates conceptually treating bags as an ADT using the so-called union representation.

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

  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 \(t : T A \rightarrow U A\) 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 application of \(t\) commutes with application of map \(f\) for all \(f\). This formalism, however, cannot be directly applied in all cases. For example toMatrix : Bag \((N, A)\) \rightarrow 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 has to be referentially transparent (i.e., purely functional). Moreover, in order to facilitate efficient and concise
implementations 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. (R3) Control and data flow should be handled in a unified way.

All of 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 which 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

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. One example is projection push down. To enable this optimization, we derive from the code that the fields to be pushed down are never accessed. Another example is filters that are e.g., applied on matrices whereas they can be pushed to the DataBags from which the matrices originate. 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 which 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 in order to execute the join, and then shuffle once again to block-partition the matrix 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 \( rowIdx \) 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:

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

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, taking into account 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.
\[ k = \left( \frac{ID}{\text{rowBlkSize}}, \frac{\text{resultColIdx}}{\text{colBlkSize}} \right) \]

where \( \text{resultColIdx} \) is the index of the column in the (now virtual) bag \( X \) and \( ID \) is the primary key of the inputs. As the schema of the join result is explicitly provided in \textit{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.

![Figure 1](image)

\( \require{cancel} \)

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

### 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 ( c <- M.rows() ) yield (mean(c))
```

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 \textit{DataBag} representation in a single pass, as we have tuple/row at-a-time processing. In a typical hash-partitioned join, the aggregate could be calculated 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 which implicitly caches data used repeatedly in iterations.
5 Concrete API

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 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).

5.1 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 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]
```

5.3 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]
```

5.4 Binary Matrix-Matrix and Matrix-Vector Operators

```python
def %*%(that: Matrix[A]): Matrix[A]
def %*%(that: Vector[A]): Vector[A]
```
5.5 Matrix self-transformations Operators

```scala
def diag(): Vector[A]

def transpose(): Matrix[A]

def row(rowIndex: Int): Vector[A]

def column(colIndex: Int): Vector[A]


def indexedRows[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]
```
6 Example Algorithms implemented in Lara

6.1 K-Means

/**
 * KMeans example implemented with the matrix abstraction of Emma
 */
object KMeansMatrix {
/**
 * @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
val centroids : Matrix[Double] = Matrix.fillColumns(n,k)(j => x.row(Random.nextInt(m)))
val 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
  centroids = 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 )

// return the trained model
newCentroids.transpose()}

// Example Algorithms implemented in Lara
// K-Means
// /*
//  * KMeans example implemented with the matrix abstraction of Emma
//  */
// object KMeansMatrix {
//  /**
//   * @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
//      centroids = 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 )
//    // return the trained model
//    newCentroids.transpose()}
// */
6.2  Linear Regression

```scala
import eu.stratosphere.emma.api.lara.{Matrix, Vector}

import scala.util.Random

/** Linear Regression example
 * Computes simple linear regression with batch gradient descent.
 * The data can be generated in two different ways given the API.
 */
object LinearRegressionGD {

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

  /** Generate training data with labels/target values
   * @param numPoints number of datapoints to generate, shape of resulting matrix x will be (numPoints,2)
   * @param bias the bias of the target values
   * @param variance variance in the target labels (noise)
   * @return pair of training data and labels/target values
   */
  def genData(numPoints: Int, bias: Double, variance: Double): (Matrix[Double], Vector[Double]) = {
    // generate data in databag comprehension style
    val m = for (i <- x.rRange; j <- x.cRange) yield {
      if (j == 0) (i, j, 1)
      else        (i, j, i)
    }
    // generate data with assignments
    val x = Matrix.zeros[Double](numPoints, 2)
    val y = Vector.zeros[Double](numPoints)
    for (i <- x.rRange /: j <- x.cRange *) {
      x(i, 0) = 1
      x(i, 1) = i
    }

    /*
     * generate data with fill
     */
    val x = Matrix.fill[Double](numPoints, 2)((i,j) => if (j == 0) 1 else i)
    val y = Vector.fill[Double](numPoints)(i => (i + bias) + Random.nextDouble() * variance)

    (x, y)
  }
}
```
7 Conclusion

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

Lara, 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).

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 which 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. Examples are projection push down (based on knowledge that fields are never accessed), as well as filters that are e.g., applied on matrices whereas they can be pushed to the DataBags from which the matrices originate. In the sequel we present more sophisticated optimizations that come from a deeper analysis of a program’s code provided in document.

Finally, the sample 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 develop scalable machine learning algorithms.
References


