PROTEUS

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

687691

D3.8 Declarative Language Finished Implementation

Lead Authors: Jeyhun Karimov and Alireza Rezaei Mahdiraji
With contributions from: Bonaventura Del Monte
Reviewer: Hamid Bouchachia (BU)

Deliverable nature:	Demonstrator (D)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	30.11.2017
Actual delivery date:	30.11.2017
Version:	0.8
Total number of pages:	20
Keywords:	Declarative language, domain specific language, streaming matrices, Lara streaming, machine learning, linear algebra

Abstract

Declarative language is one of the essential parts of PROTEUS to analyse input streaming records easily. The goal of this deliverable is to describe the finished implementation of PROTEUS declarative language for online machine learning. The dataflow systems have two main limitations in dealing with analytics, which consists of a mixture of linear algebra and relational algebra operators. One limitation is that dataflow engines are unaware of linear algebra operators, which hinders the possible optimizations. Another limitation is that low-level APIs are difficult to adopt by machine learning specialists. Previously, we introduced the Lara language which makes the system's specific aspects transparent to the user and provides more opportunities for the query compiler to optimize the resulting execution plan. In this document, we present an extension to the Lara language, which enables Lara to support scenarios containing distributed streamed matrices. In particular, we introduce distributed streaming matrix data model and operations as well as their implementation in Lara using Scala.

Executive summary

PROTEUS brings together machine learning, data, and systems scientists. However, it is non-trivial to transfer all the required knowledge among those fields on-demand because of the need of a framework and automatization to avoid the complex manual work. Previously, we proposed the Lara declarative language to merge the unify concepts from machine learning, scalable data management, and systems.

Researchers often adopt R or Matlab-like prototyping languages to write their algorithms. However, there are several limitations of this approach. First, R and Matlab do not scale up to cope with massive amounts of input data as in PROTEUS. This results in a bottleneck. Moreover, R and Matlab cannot handle streaming data in a smooth fashion. One can simulate stream processing in R by executing a sequence of R scripts. However, this can cause high latency for the end-user and this approach does not preserve the time order of events. Our approach on the other hand, combines the capabilities of stream data processing in scale, linear and relational algebra optimizations in one language.

In this deliverable, first we introduce an extension of the first Lara prototype, which supports a novel streaming matrix data model. First, we provide the theoretical background behind such streaming matrix data model. We explain the new data model, which is called Distributed Streaming Data Matrix Model. We formally define the streaming matrix data model and operations on it. The data model and specifications of matrix operations are inspired by the distributed streaming matrix data model and approximate computations of functions on such matrices.

Second, we provide the main intuition behind our implementation. We aim to support typical linear algebra operations among matrices, which are streamed from an input source. Input source can be user-defined/static/real-time. The streaming system ingests and processes data from a source. This processing is a combination of three main steps: 1) the streaming system builds those matrices from the input data (e.g., samples matrix), 2) initial computation is performed and intermediate results are materialized, and 3) the system utilizes these materialized results of previous computation step to perform the current computation incrementally. This process continues real-time processing as long as the data source provides input.

Third, we analyze the differences between the first prototype of Lara and the implementation presented in this deliverable. We adapt our computations to streaming environment. Moreover, we introduce new dynamic matrices that can adapt to streaming environment. Furthermore, we adopt incremental computation techniques to reuse intermediate results of previous computations for the upcoming computations. We perform all of the above operations transparently to the user.

Document Information

IST Project Number	687691	**Acronym**	PROTEUS
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**	D3.8	**Title**	Declarative Language Finished Implementation
Work Package	**Number**	WP3	**Title**	Scalable Architectures for both batch data and data streams

Date of Delivery	**Contractual**	M24	**Actual**	M24
Status	version 0.7		final	
Nature	demonstrator			
Dissemination level	public			

Authors (Partner)				
Responsible Author	**Name**	Alireza Rezaei Mahdiraji	**E-mail**	alireza.rm@dfki.de
	Partner	DFKI	**Phone**	+49 30 23895 6627

Abstract (for dissemination)	Declarative language is one of the essential parts of PROTEUS to analyse input streaming records easily. The goal of this deliverable is to describe the finished implementation of PROTEUS declarative language for online machine learning. The dataflow systems have two main limitations in dealing with analytics, which consists of a mixture of linear algebra and relational algebra operators. One limitation is that dataflow engines are unaware of linear algebra operators, which hinders the possible optimizations. Another limitation is that low-level APIs are difficult to adopt by machine learning specialists. Previously, we introduced the Lara language which makes the system's specific aspects transparent to the user and provides more opportunities for the query compiler to optimize the resulting execution plan. In this document, we present an extension to the Lara language, which enables Lara to support scenarios containing distributed streamed matrices. In particular, we introduce distributed streaming matrix data model and operations as well as their implementation in Lara using Scala.
Keywords	Declarative language, domain specific language, streaming matrices, Lara streaming, machine learning, linear algebra

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
2017-11-02	0.1	Alireza Rezaei Mahdiraji	First complete draft
2017-11-03	0.2	Jeyhun Karimov	Internal revision
2017-11-06	0.3	Bonaventura Del Monte	Internal revision
2017-11-07	0.4	Alireza Rezaei Mahdiraji	Some fixes and finalisation

2017-11-07	0.5	Bonaventura Del Monte	Minor improvement and finalisation
2017-11-22	0.6	Bonaventura Del Monte	Integrate Reviewer's comments
2017-11-22	0.7	Bonaventura Del Monte	Address 2nd round of Reviewer's comments
2017-11-28	0.8	Jeyhun Karimov	Address 3rd round of Reviewer's comments

Table of Contents

List of Figures

List of Tables

List of Code Snippet Listings

Abbreviations

A list of abbreviations in alphabetical order is strongly recommended. See the following example.

API: Application Programming Interface

DSL: Domain Specific Language

JIT: Just-in-Time

LASSO: Least Absolute Shrinkage and Selection Operator

1 Introduction

In Deliverable D3.7, we presented a first prototype of the declarative language *Lara*, with final syntax definition, language foundations, optimizations used and sample APIs. Lara aims to allow data scientists to define their machine learning algorithms in one engine-agnostic language. Then, Lara compiles the input program for the selected processing engine (e.g., Apache Flink). To this end, Lara insert in the program explicit parallelization constructs of the selected engine (e.g., flatMap, coProcess, and countWindow). Moreover, it enables joint optimization over both relational and linear algebra. 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 top of 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 the user program into a unified intermediate representation for joint optimization of linear and relational algebra. By leveraging on this IR, a just-in-time (JIT) compiler generates code for different execution engines (e.g., Apache Spark and Apache Flink). Lara contains three core types, namely, *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. Moreover, Lara offers Type Conversions that are natural transformations – polymorphic functions and Control-Flows. The first prototype of Lara includes its core APIs and example algorithms that show overall intuition of the language. The API closely resembles languages such as R, Matlab, and libraries such as NumPy [9]. This was done intentionally. Data science experts are traditionally accustomed to such declarative languages. Thereby, in PROTEUS we aim to make sure that data scientists, who do not have system-specific knowledge, can efficiently develop scalable machine learning analytics.

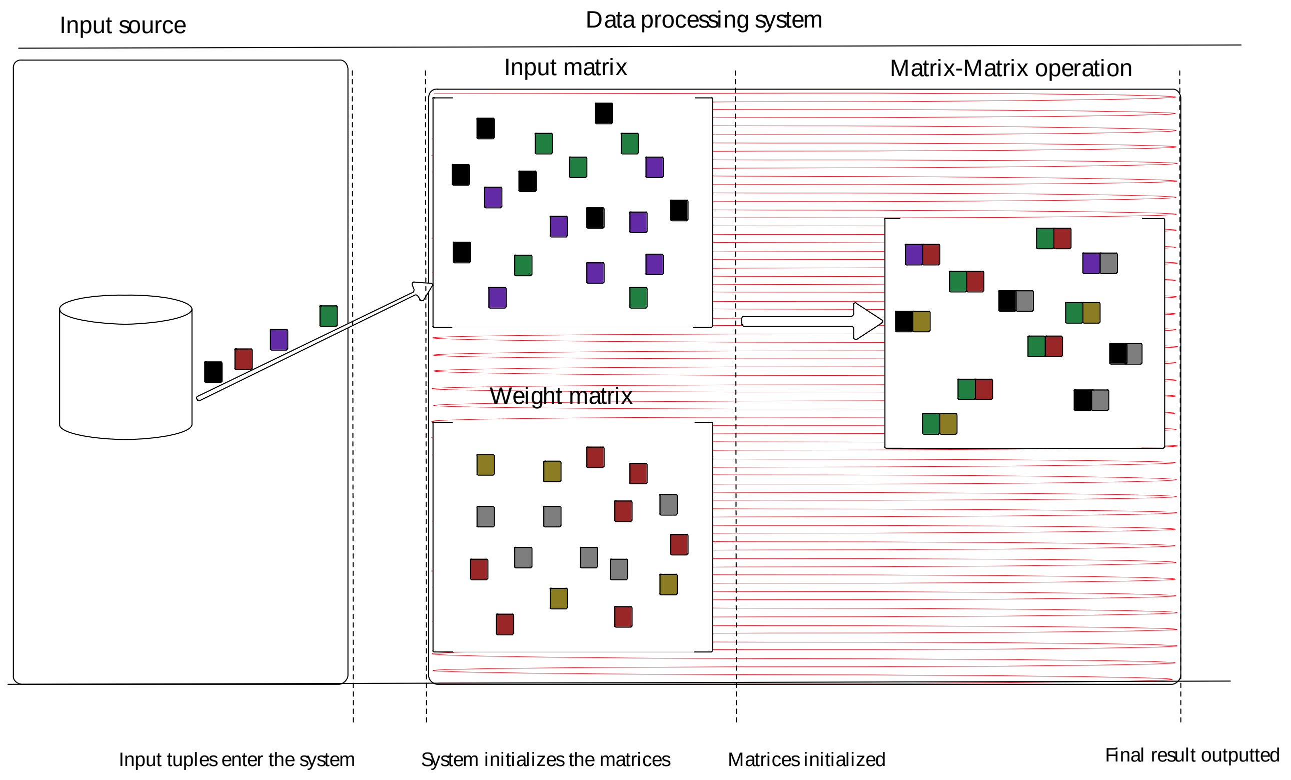

Figure 1. Illustration of how the first prototype of Lara works.

In this deliverable, we present an extension of the Lara language, which enables expressing scenarios that are known as *distributed streaming matrix* [4]. In the distributed streaming matrix scenarios, each item of the stream is a row in a matrix with fixed number of columns and possibly infinite number of rows, as the stream can be unbounded. Thus, the size of the matrix evolves over time as new items arrive. This means that the matrix has a fixed dimension only at each specific instant of time. The goal of analytics is usually to compute some function of the streaming matrix up to current state of the stream [4]. Online algorithms such as frequent direction [5] (which also has been described in Deliverable D4.2 of WP4 [6]) or online regression algorithms such as online LASSO as described in Deliverable D4.3 of WP4 [7] assume that the incoming data are rows of a growing matrix.

The proposed extension of Lara is suitable to handle stream data on which a data scientist wants to perform advanced analytics (i.e., machine learning on data-in-motion). This feature is a great candidate for the PROTEUS project as its main goal is to investigate and develop ready-to-use scalable online machine learning algorithms. In the specific industrial use case of PROTEUS, which is provided by ArcelorMittal – a consortium industrial partner, one task is to predict the degree of flatness of coil based on machinery sensors, which produce a large amount of data. Given the large number of data coming from ArcelorMittal’s sensors, it is not feasible to manually inspect deviations in flatness the coils. Therefore, one of the main aim of WP4 is to develop a method to monitor the incoming stream of sensor data and predict flatness of coils. If there is a flatness issue, a possible action would be to stop the rolling process or to automatically downgrade the quality of the coil to a less demanding client [8]. One solution for such problem is to use the online version of LASSO linear model as described in [7]. Online LASSO aims to enhance the prediction accuracy and interpretability of the statistical model by performing both variable selection and regularization [1]. As we mentioned online LASSO [7] assumes that the input streaming is a growing matrix. Moreover, distributing and parallelising the existing machine learning algorithms is another problem that needs to be addressed. In this deliverable, we introduce and extension to Lara language, which offers abstract data types for distributed streaming matrices based on DataStream API of Apache Flink. The rest of this deliverable is structured as follows: we present the streaming matrix data model in Section 2, we show a new abstract data type called StreamingMatrix in Scala and syntax definition of most important matrix operations in Section 3, and we conclude the deliverable in Section 4.

2 Distributed Streaming Matrix Data Model

In this Section, we formally define the streaming matrix data model as well as all the operations that it supports. Both the data model and those operations take into account the distributed nature of the underlying data representation of a streaming matrix. Matrix operations are based on approximate computations of functions on such matrices from [4]. In the rest of this Section, we will use the notations in Table 1.

Table 1. Notations used to describe streaming matrix data model and operations.

Variable/Symbol	Description
$\boldsymbol{A}$	A streaming matrix
$\boldsymbol{A}_{\boldsymbol{i \times n}}$	A matrix of I rows and n columns
$\boldsymbol{A}^T$	Transpose of matrix $\boldsymbol{A}$
$\boldsymbol{S}$	A matrix
$\boldsymbol{C}$	Result matrix
$\boldsymbol{Q}$	A matrix
$\boldsymbol{P}$	A matrix
$\mathbf{0}_{n \times n}$	A $n \times n$ zero matrix, i.e., matrix with all zero entries
$\boldsymbol{a}$	A vector
$\boldsymbol{x}$	A vector
$\boldsymbol{y}$	A Vector
$\boldsymbol{f}$	An arbitrary function
+	Matrix summation
*	Multiplication operator (can be scalar-to-matrix or matrix-to-matrix depending on the operands)
s	A scalar value
|	Append operator that concatenates a vector to a streaming matrix as its last row
▯	A composable function that merges the current result with the previous one

In the distributed streaming matrix scenarios, the incoming stream consists of infinite number of items and each item has *n* attributes. This basically means that each item is a row of the streaming matrix. At the beginning of the stream, the streaming matrix has no rows, which is noted as $\mathbf{A}_{0 \times n}$. The arrival of each new stream item $\boldsymbol{a} \in R^n$ (i.e., $\boldsymbol{a}$ is n-dimensional vector) leads to row-wise growth of the streaming matrix, i.e., $\boldsymbol{a}$ will be appended as a row to the current matrix *A*. We show this using the *append* operator | as follows:

- $\boldsymbol{A}_{i \times n} \mid \boldsymbol{a} = \boldsymbol{A}_{(i+1) \times n}$

For instance, after the arrival of the first stream item $\boldsymbol{a}_1 \in R^n$, the streaming matrix can be shown as:

- $\boldsymbol{A}_{0 \times n} \mid \boldsymbol{a}_1 = \boldsymbol{A}_{1 \times n}$

This means the matrix contains only one row, which is the newly arrived element $\boldsymbol{a}_1$.

Assuming at time *t* we observed *m* items of the incoming stream, the streaming matrix $\boldsymbol{A}$ is of dimension $m \times n$, i.e., $\boldsymbol{A}_{m \times n}$. The user query imposed on the streaming matrix can be thought of as computing an arbitrary function *f* on the current state of the streaming matrix, i.e., $f(\boldsymbol{A})$. Since, each new item should be processed in a streaming fashion, we need to incrementally update the query answer. This means that each new item (i.e., a row in the streaming matrix) leads to some changes to the result of the function *f*. Formally, we can represent this recursively as follow:

- $f(\boldsymbol{A}_{0\times n}) \equiv \boldsymbol{A}_{0\times n}$
- $f(\boldsymbol{A}_{j\times n}) \equiv \mathrm{f}(\boldsymbol{A}_{(j-1)\times n}) \oplus \mathrm{f}(\boldsymbol{a})$

The first expression means that the application of the function *f* on an empty streaming matrix (i.e., $\boldsymbol{A}_{0\times n}$) will output the same empty matrix. The second expression shows that one can approximate the function *f* incrementally by merging (shown using the circled plus notation $\oplus$) the previous value of *f* with application of *f* to the latest stream element.

The limitation of this approach is that the function *f* constrained to be mergeable or one should define an approximation of the function, which enables incremental computation.

In the sequel, we show that matrix operations such as multiplication in streaming scenarios can be constructed incrementally with some basic assumptions and using the append operator, defined above, for merging the results of the consecutive steps. For instance, we show formally how scalar-to-matrix and matrix-to-matrix multiplication operations, which are two core operations in linear algebra, can be formulated in such a way to be computed incrementally in online streaming scenarios.

Assuming *s* as a scalar, a $\boldsymbol{A}_{j\times n}$ streaming matrix with *j* rows and **C** as the result streaming matrix, multiplication of *s* and the matrix can be recursively defined as follows:

- $\boldsymbol{C}_0 = s * \boldsymbol{A}_{0\times n} = \boldsymbol{A}_{0\times n}$
- $\boldsymbol{C}_j = s * \boldsymbol{A}_{j\times n} = s * (\boldsymbol{A}_{(j-1)\times n} \mid \boldsymbol{a}) = (s * \boldsymbol{A}_{(j-1)\times n}) \mid (s * \boldsymbol{a}) = \boldsymbol{C}_{j-1} \mid (s * \boldsymbol{a})$

This means with arrival of each new stream item $\boldsymbol{a}$, we only need to compute $s*\boldsymbol{a}$ and append it to the previous value of **C** , i.e., $\boldsymbol{C}_{j-1} = s * \boldsymbol{A}_{(j-1)\times n}$.

Assuming S is a static matrix (i.e., a non-streaming matrix) of dimension $n\times p$, $\boldsymbol{A}_{j\times n}$ is a streaming matrix and **C** is the result streaming matrix, then the matrix multiplication of ***A*** and ***S*** can be recursively expressed as:

- $\boldsymbol{C}_0 = \boldsymbol{A}_{0\times n} * \mathbf{S}_{n\times p} = \boldsymbol{A}_{0\times n}$
- $\boldsymbol{C}_j = \boldsymbol{A}_{j\times n} * \mathbf{S}_{n\times p} = (\boldsymbol{A}_{(j-1)\times n} \mid \boldsymbol{a}) \times \mathrm{S}_{n\times p} = (\boldsymbol{A}_{(j-1)\times n} * \mathrm{S}_{n\times p}) \mid (\boldsymbol{a} * \mathbf{S}_{n\times p}) = \boldsymbol{C}_{j-1} \mid (\boldsymbol{a} * \boldsymbol{S}_{n\times p})$

This means that upon arrival of each new stream item *a*, we compute the vector to matrix multiplication $\boldsymbol{a} * \mathbf{S}_{n\times p}$ and append it to the current result of the multiplication, i.e., $\boldsymbol{C}_{j-1}=\boldsymbol{A}_{(j-1)\times n} \times \mathbf{S}_{n\times p}$.

Similarly, one can show that other matrix operations can be computed in incremental fashion as new stream items (i.e., rows of the streaming matrix) arrive.

More complex scenarios require defining other merging functions rather than the append operator to approximate the given function *f*. For instance, we try to show how this can be done for *online recursive least squares*.

Let assume that ***A*** is an $m\times n$ matrix of m rows with *n* attributes, $\boldsymbol{W}_{n\times 1}$ is the vector of coefficients and $\boldsymbol{y}_{m\times 1}$ is the vector of target variable. The least square solution for the system of equation $\boldsymbol{A}*\boldsymbol{W} = \boldsymbol{y}$ is:

- $\boldsymbol{A}^T * \boldsymbol{A} * \boldsymbol{W} = \boldsymbol{A}^T * \boldsymbol{y}$

With arrival of a new item $\boldsymbol{a}_{m+1}$ and target value y_{m+1} , all we need to do is to update the matrix $\boldsymbol{P} = \boldsymbol{A}^T * \boldsymbol{A}$ by adding the product $\boldsymbol{a}^T_{m+1} * \boldsymbol{a}_{m+1}$ to it and update $\boldsymbol{Q} = \boldsymbol{A}^T * \boldsymbol{y}$ by adding $y_{m+1} * \boldsymbol{x}^T_{m+1}$ (y_{m+1} is a scalar) Since, the for the final solution we only need the updated values of $\boldsymbol{P}$ and $\boldsymbol{Q}$, we need to keep updating them upon arrival of new stream items. In this scenario, the resulting matrix $\boldsymbol{P}$ and vector $\boldsymbol{Q}$ are static matrices (i.e., with predetermined dimension) that are computed incrementally as the streaming matrix $\boldsymbol{A}$ and the streaming vector $\boldsymbol{y}$ grow. We show how $\boldsymbol{P}$ can is computed based on what shown above. $\boldsymbol{P}$ is a function of the matrix $\boldsymbol{A}$, i.e., $\boldsymbol{P} = f(\boldsymbol{A}) = \boldsymbol{A}^T * \boldsymbol{A}$. Thus, we show how $\boldsymbol{P}$ is approximated incrementally by explaining how the function $f(\boldsymbol{A})$ is computed. To show this, we can use the recursive approach that we presented above to approximate an arbitrary function f on a streaming matrix:

- $f(\boldsymbol{A}_{0\times n}) = \boldsymbol{A}_{0\times n}{}^T * \boldsymbol{A}_{0\times n} = \boldsymbol{A}_{n\times n} = \mathbf{0}_{n\times n}$
- $f(\boldsymbol{A}_{(m+1)\times n}) = \mathbf{f}(\boldsymbol{A}_{m\times n}) + \mathbf{f}(\boldsymbol{a}_{m+1}) = \mathrm{f}(\boldsymbol{A}_{m\times n}) + (\boldsymbol{a}^T{}_{m+1} * \boldsymbol{a}_{m+1})$

Basically, the first statement states that the application of the function f on an empty streaming matrix outputs a matrix of dimensions $n\times n$. To be consistent with least square computation, one can assume this is a zero matrix, i.e, a matrix with all zero entries, which is shown as $\mathbf{0}_{n\times n}$.

The second statement declares that the value of f after arrival of the $(m+1)$-th element can be estimated by summing its previous value and $\boldsymbol{a}^T{}_{m+1} * \boldsymbol{a}_{m+1}$. Note that here the merging operation is the known matrix summation.

Similarly, one can show how $\boldsymbol{Q}$ can be incrementally computed as the stream of new rows of $\boldsymbol{A}$ and y values are ingested in the processing system.

3 Implementations of Distributed Streaming Matrix

Similar to Lara, we implement streaming matrix extension of Lara in Scala [3]. Scala merges two main programming paradigms, i.e., object-oriented and functional programming in a multi-paradigm programming language. This is the main reason behind the choice of Scala as programming language for in our implementation. Moreover, Scala expresses common programming patterns in a much more concise manner than other object-oriented languages such as Java. Another reason is that Flink supports Scala in DataSet and DataStream APIs.
To adapt Lara to streaming environment, it is necessary to develop dynamic data types that model streaming matrices and vectors. As we explained in details in Section 2, streaming matrices grow with the arrival of new items of the input data stream and the goal is to approximate some computation expressed by a function on such matrix in an incremental fashion.

To meet with a broad range of needs stemming from both machine learning to data processing, it is essential to avoid specific use-case based implementations but support generic types. Thus, we introduce generic matrix data type. Scala generics, known as parametric polymorphism, allows for various declarations and definitions to take type parameters. We assign type parameters to methods, classes, traits, type aliases, and type members.

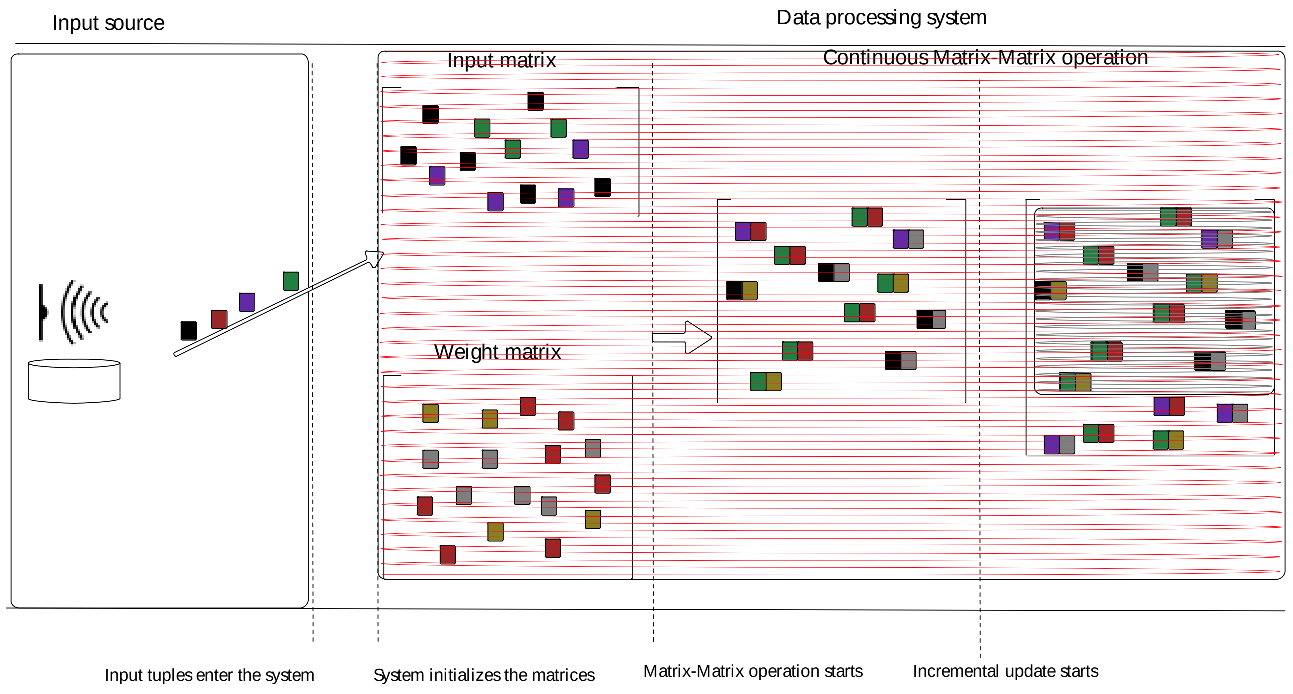
Input source
Data processing system
Input matrix
Continuous Matrix-Matrix operation
Weight matrix
Input tuples enter the system
System initializes the matrices
Matrix-Matrix operation starts
Incremental update starts

Figure 2. Illustration of continuous matrix-to-matrix operation based on streaming matrix data model.

Figure 2 shows the main intuition behind our implementation. Initially, data are read from input source (blue part). Afterwards, data are ingested to streaming system (red part). The processing is a combination of three main steps. First, the streaming system builds the user-defined matrixes. For instance, a ML algorithm usually requires two matrices: one containing the input samples and one containing the parameters (i.e., weights) of the underlying ML model. The matrix of input sample is read from the input stream, whereas the parameters of the ML model are usually initialized using some random distribution or using the parameters of a previously trained model. Second, initial computation is performed and intermediate results are materialized. Finally, the system utilizes the materialized results of previous computation step to perform the current computation incrementally. This process continues real-time processing as long as data source provides input.

We extend Lara with streaming matrix data model. In the following, we explain the main features of the streaming extension and its differences to the first prototype of Lara:

- As we can see in Figure 2, the proposed extension to Lara adopts the previously defined abstractions (see Deliverable 3.7) for streaming scenarios. Thus, it is more generic implementation than batch because batch processing can be thought as a special case of stream data processing where input data are bounded.
- The current version of Lara accepts input data both from batch and online data sources. We adapt our implementation to the PROTEUS use-case, which is tightly coupled with online and real-time input sources.
- We introduce streaming matrix in our implementation. As we can see in Figure 2, the input matrix never ends. It is updated regularly and is kept as a global big matrix. Thus, rather than working with fixed sized matrices (as it was in the batch version of Lara), we adapt our implementation to online scenarios in which the input matrix size changes dynamically.
- We incrementally compute user-defined operations. This is shown in the right part of the pipeline in Figure 2. The grey part of the matrix is used as intermediate result from the previous computations. The rest is computed and the result is merged with intermediate materialized results. This approach is essential in streaming scenarios and computation heavy environments.
- We perform all of the above implementation and optimizations transparently to the end user. As a result, the user can focus on the algorithm and the language will take care of distribution, parallelization, and efficiency.

In the following, we show how we implement the streaming matrix data model and operations in Scala based on Flink DataStream API.

```
class StreamingMatrix(
     @transient private[lara] val ds: DataStream[Array[Double]],
     private[lara] val numCols: Int,
     private[lara] val numRows: Int
     ) extends Serializable {…}
```

Listing 1. Streaming matrix abstraction on top of DataStream API.

Listing 1 shows the Distributed Streaming Matrix data model represented as an abstract data type in Scala. We build streaming matrix incrementally with Array of doubles. At any time, we keep column and row size of our abstraction as it is dynamically changing data structure.

```
def +(that: Double): StreamingMatrix = mapScalar(that, _ + _)
def -(that: Double): StreamingMatrix = mapScalar(that, _ - _)
def *(that: Double): StreamingMatrix = mapScalar(that, _ * _)
def /(that: Double): StreamingMatrix = mapScalar(that, _ / _)
```

Listing 2. Scalar operation methods with streaming matrix.

Listing 2 shows supported scalar operations of the streaming matrix data type. We utilize mapScalar() method to hide the implementation abstraction from the end user.

```
def +(that: StreamingMatrix): StreamingMatrix = transformMatrix(that, _ +:+ _)
def -(that: StreamingMatrix): StreamingMatrix = transformMatrix(that, _ -:- _)
def *(that: StreamingMatrix): StreamingMatrix = transformMatrix(that, _ *:* _)
def /(that: StreamingMatrix): StreamingMatrix = transformMatrix(that, _ /:/ _)
def %*%(that: StreamingMatrix): StreamingMatrix = transformMatrix(that, _ * _)
```

Listing 3. Matrix-matrix operation methods with streaming matrix.

Listing 3 shows the matrix-to-matrix operation methods for the streaming matrix data type. Here, we utilize transformMatrix() method to perform all the implementation details mentioned above and hide them from the user.

4 Conclusions

In this deliverable, we provide the details of our finished implementation of the declarative language. The contributions of this deliverable are the following:

- We analyse the differences between the first prototype of Lara and its finished implementation,
- We adapt the first prototype of Lara to distributed streaming environment, which is also compatible with the main use-case of PROTEUS, i.e., processing large amounts of data in a real-time fashion,
- We adapt an existing distributed streaming data model definition and use it to formally define scenarios and operations involving streaming matrices,
- We explain the implementation details and the intuition behind our incremental computation semantics.

References

[1] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society. Series B (Methodological), 267-288.

[2] https://cran.r-project.org/web/packages/glmnet/glmnet.pdf

[3] https://www.scala-lang.org/

[4] Ghashami, Mina, Jeff M. Phillips, and Feifei Li. Continuous matrix approximation on distributed data. Proceedings of the VLDB Endowment 7.10 (2014): 809-820.

[5] Ghashami, Mina, et al. Frequent directions: Simple and deterministic matrix sketching. SIAM Journal on Computing 45.5 (2016): 1762-1792.

[6] D4.2 Basic Scalable Streaming Algorithms

[7] D4.3 Novel Scalable Online Machine Learning Algorithms for Data Streams

[8] D2.8 First prototype (V1)

[9] http://www.numpy.org/