D3.11 Optimizer Finished Implementation

Lead Author: Bonaventura Del Monte
With contributions from: Jeyhun Karimov, Alireza Rezaei Mahdiraji
Reviewer: Javier De Matias Bejarano (TREE)

<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>Contractual delivery date:</td>
<td>31.08.2018</td>
</tr>
<tr>
<td>Actual delivery date:</td>
<td>31.08.2018</td>
</tr>
<tr>
<td>Version:</td>
<td>1.0</td>
</tr>
<tr>
<td>Total number of pages:</td>
<td>20</td>
</tr>
<tr>
<td>Keywords:</td>
<td>Optimization, Domain Specific Language, Machine Learning, Dataflow Engine, Operator Fusion</td>
</tr>
</tbody>
</table>
Abstract

In this deliverable, we describe our first prototype of the PROTEUS optimizer. One of the main objectives of PROTEUS is to provide the end-users with a Domain Specific Language (DSL) that allows them to define Machine Learning algorithms on data streams. We proposed Lara as DSL for linear algebra and relational operations on streams (for details on Lara see D3.4 – D3.8). Our proposed language maps each linear algebra operation to one specific high-order operator of the underlying execution dataflow engine (e.g., Apache Flink). However, the language requires an extra module to perform holistic optimization on the sequence of high-order operators, which are shipped to the execution engine for the actual processing. To this end, we introduced the PROTEUS optimizer in D3.10. The task of the optimizer is to analyze the logical execution plan of a streaming query (containing Machine Learning methods) and perform non-trivial optimizations tailored to the domain specific language. In particular, the PROTEUS optimizer performs operator fusion for Sum-Product operators. This optimization results in fewer intermediate results and resource efficiency. We target this class of optimizations because almost all the PROTEUS workloads contain machine learning methods, which rely on linear algebra operations.

This enables better resource management of the underlying execution engine as it results in fewer operator being scheduled for execution.

In this deliverable, we assess the capabilities of the optimizer through benchmarks.

[End of abstract]
Executive summary

In this deliverable, we describe a first prototype of the PROTEUS optimizer. One of the main objectives of PROTEUS is to provide the end-users with a Domain Specific Language that allows them to define Machine Learning algorithms on data streams. To that end, we proposed Lara as Domain Specific Language for linear algebra operations on streams (see D3.4 – D3.8). Our proposed language maps each linear algebra in one specific high-order operator of the underlying execution engine (e.g., Apache Flink). However, the language requires an extra module to perform holistic optimization on the sequence of high-order operators, which are shipped to the execution engine for the actual processing. To this end, we introduce the PROTEUS optimizer, a module that analyzes the logical execution plan of a streaming query and perform non-trivial optimizations tailored to the domain specific language; namely, it performs operator fusion for Sum-Product operators. This enables better resource management of the underlying execution engine as it results in fewer operator being scheduled for execution.

In this document, we demonstrate how a query plan produced by our PROTEUS optimizer is similar to the hand-coded execution plan developed in D4.5. Furthermore, we show how the query plan generated by our PROTEUS optimizer leads to performance close to the ones of the hand-coded execution plan. As a result, the use of our Lara DSL in combination with our PROTEUS optimizer drastically reduces the amount of expertise that the end-user requires to write analytics pipelines, thus, reducing the time to knowledge.
In this deliverable, we describe our first prototype of the PROTEUS optimizer. One of the main objectives of PROTEUS is to provide the end-users with a Domain Specific Language (DSL) that allows them to define Machine Learning algorithms on data streams. We proposed Lara as DSL for linear algebra and relational operations on streams (for details on Lara see D3.4 – D3.8). Our proposed language maps each linear algebra operation to one specific high-order operator of the underlying execution dataflow engine (e.g., Apache Flink). However, the language requires an extra module to perform holistic optimization on the sequence of high-order operators, which are shipped to the execution engine for the actual processing. To this end, we introduced the PROTEUS optimizer in D3.10. The task of the optimizer is to analyze the logical execution plan of a streaming query (containing Machine Learning methods) and perform non-trivial optimizations tailored to the domain specific language. In particular, the PROTEUS optimizer performs operator fusion for Sum-Product operators. This optimization results in fewer intermediate results and resource efficiency. We target this class of optimizations because almost all the PROTEUS workloads contain machine learning methods, which rely on linear algebra operations.

This enables better resource management of the underlying execution engine as it results in fewer operator being scheduled for execution.

In this deliverable, we assess the capabilities of the optimizer through benchmarks.

Keywords
Optimization, Domain Specific Language, Machine Learning, Dataflow Engine, Operator Fusion
<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Reviewer</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.08.2018</td>
<td>0.4</td>
<td>Javier De Matias Bejarano</td>
<td>First external Review</td>
</tr>
<tr>
<td>27.08.2018</td>
<td>1.0</td>
<td>Bonaventura Del Monte</td>
<td>Address reviewer’s comments</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
1 Introduction............................................................................................................................ 9
2 The PROTEUS Optimizer .................................................................................................... 10
3 Benchmarks .......................................................................................................................... 12
   3.1 Benchmark description .................................................................................................. 12
   3.1.1 Benchmark model ..................................................................................................... 12
   3.1.2 Data model ................................................................................................................ 14
   3.1.3 Use case .................................................................................................................... 15
3.2 KPIs .................................................................................................................................. 15
3.3 Code Complexity Comparison ......................................................................................... 16
   3.3.1 The hand-coded query plan of LASSO ..................................................................... 16
   3.3.2 LASSO implementation in Lara DSL ....................................................................... 18
3.4 Execution Plan Comparison ............................................................................................. 18
3.5 Performance comparison .................................................................................................. 19
4 Conclusions .......................................................................................................................... 20
References ............................................................................................................................... 21
List of figures and/or list of tables

Figure 1 Simple execution plan for X * W − Y ................................................................. 11
Figure 2 Optimized execution plan for X * W − Y ................................................................. 11
Figure 3. Closed benchmark model ......................................................................................... 12
Figure 4. Open benchmark model ......................................................................................... 13
Figure 5. Partly open benchmark model ............................................................................... 13
Figure 6. Tuple fields for SensorMeasurement2D .................................................................. 14
Figure 7. Tuple fields for SensorMeasurement1D .................................................................. 14
Figure 8 Lasso Hand-Coded Execution Plan ....................................................................... 19
Figure 9 Optimizer-generated Execution Plan ...................................................................... 19
1 Introduction

One of the main objectives of PROTEUS is to provide the end-users with a Domain Specific Language (DSL) that allows them to define Machine Learning algorithms on data streams. We proposed Lara as DSL for linear algebra and relational algebra operations on streams (for details on Lara see D3.4 – D3.8). The Lara language maps each linear algebra operation to one specific high-order operator of the underlying execution dataflow engine (e.g., Apache Flink). However, our language compiler requires an extra component to perform holistic optimization on the sequence of high-order operators, which are shipped to the execution engine for the actual processing. To this end, we introduce the PROTEUS optimizer in this deliverable. Our PROTEUS Optimizer is designed to analyze the logical execution plan of a streaming query (containing Machine Learning methods) and to perform not-trivial optimizations. These optimizations are tailored to the domain specific language and thus for analytics workloads containing online machine learning methods. In particular, the PROTEUS optimizer performs operator fusion for Sum-Product operators. This optimization results in fewer intermediate results and resource efficiency. We target this class of optimizations because almost all the PROTEUS workloads contain machine learning methods, which rely on linear algebra operations. This enables better resource management of the underlying execution engine as it results in fewer operators being scheduled for execution.

In this deliverable, we show how a query plan produced by our PROTEUS optimizer is similar to the hand-coded execution plan developed in D4.5. Furthermore, we show also how the query plan generated by our PROTEUS optimizer leads to performance close to the ones of the hand-coded execution plan.

This document is structured as follows: we present a brief overview of our PROTEUS optimizer in Section 2 and our benchmarks in Section 3. Then, we provide the conclusion in Section 4.
2 The PROTEUS Optimizer

The PROTEUS Optimizer allows the end-user to define online machine learning algorithms using a domain specific language that does not require the end-user to know system-specific constructs.

In the below snippet, which we also presented in D3.10, we show how we can abstract the complexity of the feedback loop and we use an online Lasso formulation close to the one used in the PROTEUS SOLMA library (see D4.3), which was used in the Flink example and we let the end-user define only:

1. how to compute the local parameter of the model given a current model and a minibatch of points coming from the input stream and output a prediction
2. how to merge two models
3. how to update the model given the true target vector for a given point

```scala
1. val env = ...
2. val trainingStream = env.addSource(...).toMatrix
3. val initialModel = env.addSource(LassoModelRandomInitializer()).toMatrix
4. withFeedback(trainingStream, initialModel) {
5.  (X, model) => {
6.    val d = diag(DenseMatrix(X.colsNum)(sqrt(abs(model.gamma))))
7.    val A = model.A + X %*% X.t + inv(d)
8.    val prediction = model.b.t %*% inv(A) %*% X
9.    (prediction, (A, gamma))
10.  },
11.  (currModelA, currModelB) => currModelA + currModelB,
12.  (currModel, Y, X) => {
13.    val l = Y %*% X
14.    currModel.b += l
15.  }
16. }
```

1. Online LASSO implemented in PROTEUS language

To have this high-level interface, we need to obtain a holistic view on the overall set of operations we want to perform on the data stream(s). To this end, we extended the language developed in T3.3 and its underlying Intermediate Representation (IR) to support operations among streaming matrices and translate the execution flow in a streaming execution plan with feedback loops. This introduces a secondary challenge that we had to solve in this deliverable, i.e., generating the user defined functions given an arbitrary linear combination of streaming matrices. Consider the following formula, which is a common subexpression present in any General Linear Model:

\[ X \times W - y \]

where \( x \) is a matrix of size \((n, m)\), \( W \) is a matrix of size \((m, p)\), \( y \) is a matrix of size \((n, p)\), \(*\) is the matrix multiplication operator, and \(-\) is the subtraction operator among two matrices of equal size. We model each matrix as a streaming matrix that are ingested asynchronously in our processing engine. The system receives the matrix as a stream of rows of size \( m \) and triggers the computation only when \( n \) rows have been ingested. Therefore, we assume we want to perform the above operations when we receive \( n \) new rows for each of the two streaming matrices \( x \) and \( y \). A straightforward way to implement this in an execution engine such as Apache Flink is to use 2 count-window operations to materialize the \( n \) new items and then connect the stream of count-windows coming from input \( x \) to the stream of count-windows coming from the streaming matrix \( W \). Then, perform a matrix multiplication within a `coFlatMap` and then connect the stream containing the result of the `coFlatMap` to the one containing the materialized count-window of the streaming matrix \( y \). Then, compute the subtraction using the intermediate values and the last \( n \) items coming from \( y \).
The problem with this approach is that it requires at least 2p threads for the window operators (with p being the parallelism of the window operators) and 2q threads for the coFlatMaps (with q the parallelism of the coFlatMap operator). As we cannot omit the window operators, a potential optimization is to union the three streams and perform the multiplication and the subtraction in a single flatMap operator, which runs with parallelism of p. The challenge with this approach is to detect a pattern as operand1 * operand2 + operand3 and generate specialized code for the flatMap operator containing the fused math operations.

To detect those types of patterns, we leverage our Intermediate Representation format to holistically inspect the user code. Our Intermediate Representation allows for pattern matching and by exploiting such feature, our PROTEUS optimizer can fuse the three math operators into a single Flink flatMap operators.
3 Benchmarks

In this section, we aim to assess the capabilities of our PROTEUS Optimizer through benchmarking. We aim to run a simpler version of the PROTEUS Validation Scenario and we aim to show:

1. The PROTEUS Optimizer produces an execution plan for the Online LASSO method that is equal to the hand-optimized execution plan of the same algorithm presented in D4.5
2. The performance of the Online LASSO method compiled through the PROTEUS Optimizer are close in terms of throughput and latency to the performance of the hand-optimized execution plan of the same algorithm presented in D4.5

3.1 Benchmark description

In this section, we provide the overall intuition behind our benchmark. Firstly, we provide the main foundations of performance evaluation methods in the scope of PROTEUS use-case. Secondly, we discuss data characteristics provided by ArcelorMittal. Thirdly, we describe the main use-case for our benchmark. Finally, we provide main metrics we adopt in this benchmark and give their definition.

3.1.1 Benchmark model

Benchmarking is an important topic in database and systems community. The main challenge for researchers is to setup an experiment so that the system under test is accurately represented with all important features. One feature that contributes to accurate representation of underlying system under test is request arrival semantics to system under test. There are three main models for request arrival to system under test [3]. Below we elaborate the existing benchmark models and analyze which model is appropriate for our benchmark.

The first model is called closed model. As we can see from Figure 3, in a closed model new requests for system under test is triggered once the previous is completed. The new request might also have initiated after some “think” time. The model accumulates all the requests in the queue. As we can see in a closed model input requests are provided in a synchronized manner such that previous input request is blocking the later one.

As an example, we consider website in a production environment. In a closed system it is assumed that website has fixed number of users for an infinite time duration. Let the number of users be N. Each user repeatedly performs mainly two steps. Firstly, she submits a job, being a request for some resource from website. Secondly, the user receives respective response from the website. The new request might be initiated...
immediately after some time delay. Let \( N_{\text{think}} \) be the number of users thinking for some time interval before initiating a new request, \( N_{\text{run}} \) be the number of users running queries in the system under test, and \( N_{\text{system}} \) be the number of users queued to run some requests in the system under test. Then, \( N_{\text{think}} + N_{\text{system}} + N_{\text{run}} = N \).

This model is eligible for scenarios where the overall job pipeline includes blocking tasks. For example, if we use batch processing system for training our machine learning model and stream data processing model for prediction, then those two systems need to be synchronized in a closed benchmark model. However, we adopt hybrid operator and avoid extra batch processing system by implementing batch and streaming operators on top of streaming system.

![Figure 4. Open benchmark model](image)

The second model is called open model. As we can see from Figure 4, in an open model, new requests for system under test is completely uncorrelated with completion of previous request. The model accumulates all the requests in the queue. As we can see in an open model input requests can be processed in an asynchronized manner such that previous input request is not blocking the latter one.

Considering the website example, we describe above, in an open model we relax the assumption of fixed number of website users. While in a closed model we have \( N \) number of users for an infinite time duration, in an open model, the number of users might range from 0 to infinity. Each user initiates a query to the website, waits for its response and leaves. As a result, the completion of the one user query does not trigger the request of another user. The new request is triggered once a new user opens the website and initiates some request.

![Figure 5. Partly open benchmark model](image)
The third model is called partly open model. As we can see from Figure 5, this model lies between open and close model. Let’s consider the website example provided above. In partly open model, new users arrive to the website based on exiting semantics of existing users. Let $p$ be the probability of user staying in the system after initiating a query to the website. Then, in a partly open model a new user is included to the set of active users with probability $1-p$.

We chose the open model for our benchmark. The main intuition is that the hybrid capabilities of PROTEUS engine is built on top of streaming engine and benefits a full set of streaming operations. For example, for Lasso we use some amount of input data for training the model. As a result, we adopt only one system, meaning stream data processing system, and avoid an extra system, batch processing system for the training part. If we would choose third party system for batch processing (training the model), then we would adopt closed or partly open model.

### 3.1.2 Data model

PROTEUS features unique data set provided by ArcelorMittal. In general, we split the dataset into two streams:

- Measurements: the source input measurements stream
- Flatness: the source input flatness stream

Each stream contains two kinds of tuples:

- SensorMeasurement2D
- SensorMeasurement1D

<table>
<thead>
<tr>
<th>coilId</th>
<th>x</th>
<th>y</th>
<th>slice</th>
<th>data</th>
</tr>
</thead>
</table>

**Figure 6. Tuple fields for SensorMeasurement2D**

Figure 6 shows the structure for SensorMeasurement2D tuples. Below we explain fields of the tuple:

1. coilId: identification field of integer type for coil. This field is useful to parallelize the data processing topology by partitioning streams.
2. x: the coil’s x value in a 2D representation
3. y: the coil’s y value in a 2D representation
4. slice: variable ID used for measurements
5. data: variable value used for measurements

<table>
<thead>
<tr>
<th>coilId</th>
<th>x</th>
<th>slice</th>
<th>data</th>
</tr>
</thead>
</table>

**Figure 7. Tuple fields for SensorMeasurement1D**

Figure 7 shows the structure for SensorMeasurement1D tuple. As we can see the tuple structure is exactly the same with SensorMeasurement2D, but the former does not include field y.
3.1.3 Use case

We adopt Lasso for our use-case in this benchmark. The main reason to choose Lasso is that it is well suited for online machine learning scenarios. We provide delayed feedback to our Lasso model to train existing model in a real-time fashion.

Another reason for adopting Lasso is that it is well-suited for sparse matrices. Sparse matrices are attractive for machine learning frameworks, especially in breeze [4], because they can be stored using much less space with the help of space-saving methods. Moreover, sparse matrices are more interpretable than dense vectors. With Lasso regularization, we can easily handle sparse vectors.

Yet another reason for adopting Lasso is that we deal with big data. Once the entire data set cannot be loaded into the memory or it is costly to train the model with large amounts of data, we can easily stream the input data to Lasso and without prior blocking operations.

Another reason for adopting Lasso is that the data set provided by ArcelorMittal has highly correlated features. Firstly, it selects a random feature among others, which might be highly correlated. Secondly, it sets the coefficients of the other features to zero. The interesting part is that the chosen variable changes based on the real-time parameters, which triggers changes in the model, too. This is a desirable feature especially for online machine learning scenarios.

3.2 KPIs

In this section, we provide main metrics we adopt for our benchmark. We concentrate on latency in our performance evaluations. Below we explain the intuition behind not choosing throughput as a separate metric in our evaluations.

- When throughput is chosen as a separate metric, it is variable controlled and edited by benchmark. However, in our case, throughput is not a variable, it is constant and transparent to benchmark framework. The reason is that we simulate data set as real-world streaming data. To play with throughput, we would need to ingest all the data without obeying their actual timestamps, which in turn would violate the effectiveness of our benchmark and use-case.
- One argument might be to measure throughput while running experiment and consider it as a separate parameter. However, as we analyze different portions of data, starting our experiments from different offsets of Apache Kafka, we experience different throughputs. As a result, we experience non-constant and non-controllable throughput.

As a result, we configure our broker system to provide max throughput and conduct our experiments to measure latency.

Our latency definition is end-to-end latency. We use average end-to-end latency for our measurement. Below we explain the main intuition why we avoid using tuple-based latency, which is the latency for each tuple.

- In our use-case it is non-trivial to compute the latency per each tuple because there is no clear semantics to merge tuples with different source and possibly different timestamps into a tuple with a single timestamp. In Lasso, we acquire two main streaming sources: hybrid operator source and prediction operator source. Prediction operators also have two streaming sources, flatness stream and measurement stream. As a result, once we aggregate multiple tuples from multiple sources and possibly with different timestamps we will need to keep every tuple’s timestamp in the resulting tuple, which is not efficient.
- We want to treat our use-case part as blackbox. Because it is our system under test, by definition we need to keep it as transparent as possible. If we adopt tuple-based latency, then we might need to modify the internal semantics of the algorithm which might affect the overall performance.
Therefore, we use average latency. Let \( S \) be the overall number of tuples (which contains \( S^H \) the input for hybrid operator and \( S^P \) the input for real-time prediction operator) and \( T(S) \) be the time needed to process \( S \). Then, the average end-to-end latency is defined in Equation 1:

\[
L_{avg} = \frac{T(S)}{S} \tag{1}
\]

### 3.3 Code Complexity Comparison

In the following we show two code snippets: the first contains our hand-coded LASSO implementation developed in D4.5, whereas the second snippet contains a LASSO implementation implemented using our Lara DSL.

#### 3.3.1 The hand-coded query plan of LASSO

We assume that our program receives a stream of labelled data points and also uses a system-specific aspect to broadcast the updated model to the workers. In particular, we use a feedback edge in the dataflow graph in order to merge all the gradients produced by the different workers and broadcast the updated model to all the workers. The overall implementation result in a very cumbersome-to-follow block of code where language specific constructs (e.g., lambdas, Options, Either) are mixed with system-specific features (e.g., iteration, connected streams). We highlight that the below hand-crafted code snippet results in an optimal streaming execution plan when executed on Apache Flink.

```scala
1. val env = ...
2. val trainingStream = env.addSource(...).map(Left(_))
3. val initialWeights = env.addSource(WeightRandomInitializer()).map(Right(_))
4. def stepFunc(workerIn: ConnectedStreams[Matrix, Matrix]):
5. (DataStream[Matrix], DataStream[Either[Matrix, Matrix]]) = {
6.   val worker = workerIn.flatMap(new RichCoFlatMapFunction[Matrix, Matrix, Either[Matrix, Matrix]] { 
7.     @transient var stagedBatches = _
8.     @transient var currentModel = _
9.     override def open(parameters: Configuration): Unit = {
10.       stagedBatches = new mutable.Queue[Matrix]() currentModel = None
11.     } // incoming answer from PS
12.     override def flatMap2(newModel: Matrix, out: Collector[Either[Either[(Matrix, Double), Matrix], Matrix]]): Unit = {
13.       override def flatMap1(dataOrInitialModel: Either[Matrix, Matrix], out: Collector[Either[(Matrix, Double), Matrix]]): Unit = {
14.         dataOrInitialModel match {
15.           case Left(data) => {
16.             if (data.isLabelled) {
17.               currentModel match {
18.                 case Some(model) => {
19.                   val diff = miniBatch % * % model - Y
20.                   val loss = diff * diff / 2.0
21.                   val newGradient = diff * data + oldGradient
22.                   out.collect(Left(Left(newGradient, loss)))
23.                 }
24.                 case _ => stagedBatches.add(miniBatch)
25.               }
26.             } else {
27.               // do prediction here wrapping it with Right
28.             }
29.           }
30.           case Right(model) => {
31.             currentModel = Some(model) out.collect(Left(Right(model)))
32.           }
33.         }
34.       }
35.     }
36.   wOut = worker.flatMap(x => x match {
```

687691
37.     case Right(out) => Some(out)
38.     case _ => None
39.   }}.setParallelism(workerParallelism)
40.   val ps = worker.flatMap(x => x match {
41.     case Left(workerOut) => Some(workerOut)
42.     case _ => None
43.   })
44.   .setParallelism(workerParallelism)
45.   .partitionCustom(new Partitioner[Int]() {
46.     override def partition(key: Int, numPartitions: Int): Int = key % numPartitions
47.   }, paramPartitioner).flatMap{
48.     new RichFlatMapFunction[Either[(Matrix, Double), Matrix], Either[Matrix, Matrix]]{@
49.         transient val currentModel = _ override
50.         def open(p: Configuration): Unit = {
51.             super.open(p)
52.             currentModel = None
53.         }
54.         override def flatMap(msg: Either[(Matrix, Double), Matrix],
55.             out: Collector[Either[Matrix, Matrix]]): Unit = {
56.             msg match {
57.                 case Left(t) => {
58.                     val tmp = currentModel.get * t._1
59.                     val stepSizeCurrentEpoch = stepSize
60.                     val W = stepSizeCurrentEpoch * grad + (tmp / 2)
61.                     val shrinkageVal = regP * stepSizeCurrentEpoch
62.                     W = W foreach {
63.                         value => signum(value) * max(0.0, abs(value) - shrinkageVal)
64.                     }
65.                     out.collect(W)
66.                 }
67.                 case Right(model) => currentModel = Some(model)
68.             }
69.         }
70.         .setParallelism(workerParallelism)
71.         val psToWorker = ps.flatMap(_ match {
72.             case Left(x) => Some(x)
73.             case _ => None
74.         }).setParallelism(workerParallelism) // TODO avoid this empty map?
75.         .map(x => x)
76.         .setParallelism(workerParallelism)
77.         .partitionCustom(new Partitioner[Int]() {
78.             override def partition(key: Int, numPartitions: Int): Int = {
79.                 if (0 <= key && key < numPartitions) {
80.                     key
81.                 } else {
82.                     throw new RuntimeException("Pull answer key should be the partition ID itself!")
83.                 }
84.             }
85.         }, wInPartition)
86.         val psToOut = ps.flatMap(_ match {
87.             case Right(x) => Some(x)
88.             case _ => None
89.         })
90.         .setParallelism(psParallelism)
91.         val wOutEither: DataStream[Either[Either[(Matrix, Double), Matrix], Matrix], Matrix] =
92.             wOut.forward.map(x => Left(x))
93.         val psOutEither: DataStream[Either[Matrix, Matrix]] =
94.             psToOut.forward.map(x => Right(x))(
95.             psToWorker,
96.             wOutEither
97.         .setParallelism(workerParallelism)
98.         .union(psOutEither.setParallelism(psParallelism))
99.     })
100.     trainingStream
101.     .countWindow(miniBatchSize)
3.3.2 LASSO implementation in Lara DSL

```scala
17. val env = ...
18. val trainingStream = env.addSource(...).toMatrix
19. val initialModel = env.addSource(LassoModelRandomInitializer()).toMatrix
20. withFeedback(trainingStream, initialModel) {
21.   (X, model) => {
22.     val d = diag(DenseMatrix(X.colsNum)(sqrt(abs(model.gamma))))
23.     val A = model.A + X %*% X.t + inv(d)
24.     val prediction = model.b.t %*% inv(A) %*% X
25.     (prediction, (A, gamma))
26.   },
27.   (currModelA, currModelB) => currModelA + currModelB,
28.   (currModel, Y, X) => {
29.     val l = Y %*% X
30.     currModel.b += l
31.   }
32. }
```

In the above snippet, we abstract the complexity of the feedback loop and we use an online Lasso formulation close to the one used in the PROTEUS SOLMA library (see D4.5), which was used in the Flink example and we let the end-user define only:

1. how to compute the local parameter of the model given a current model and a minibatch of points coming from the input stream and output a prediction
2. how to merge two models
3. how to update the model given the true target vector for a given point

3.4 Execution Plan Comparison

The two query execution plan are almost identical. There are only two main differences:

1. The hand-coded version of LASSO does not require loading an initial model from some storage. However, the optimizer is meant for the generic case and it will inject extra operators in the optimized query plan to deal with model preloading. See operators #10, #12, #13 in Figure 9.
2. The hand-coded version of LASSO produced in D4.5 is tuned to the specific use-case of ArcelorMittal. This means that that version contains some extra filtering that reduces the amount of operations performed by the streaming pipeline. See operators #31 and #32 in Figure 8 and the lack of operators #15 and #17 that are present in Figure 9.
3.5 Performance comparison

We conduct our experiments on a cluster with 4 nodes. We run Flink on 2 of them, whereas one node is dedicated to Apache Kafka, and one node runs the Proteus data generator. Each node is equipped with Intel Xeon CPU E5530 2.40GHz with 16 cores and 32GB of RAM. PROTEUS framework simulates the input data set, thus, the overall runtime takes several days. The reason is that the input data set is derived from real production environment running for several days.

<table>
<thead>
<tr>
<th></th>
<th>2-nodes (Optimizer)</th>
<th>2-node (Hand-Written)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Latency</td>
<td>0.64</td>
<td>0.51</td>
</tr>
<tr>
<td>Min Latency</td>
<td>0.54</td>
<td>0.43</td>
</tr>
<tr>
<td>Max Latency</td>
<td>1.12</td>
<td>1.31</td>
</tr>
<tr>
<td>Percentiles (90, 95, 99)</td>
<td>(0.91, 0.93, 1.06)</td>
<td>(0.91, 0.98, 1.04)</td>
</tr>
</tbody>
</table>

Table 1 Latency Measurements (the lower, the better)

In Table 1, we show the end-to-end latency. As we can see the average latency is in the order of milliseconds and as we scale our topology the latency measurements improve. The max latency is high when compared with average latency. The high max latency is also related with system warmup at the beginning of the experiment.
4 Conclusions

In this deliverable, we show how a query plan produced by our PROTEUS optimizer is similar to the hand-coded execution plan developed in D4.5. Furthermore, we show also how the query plan generated by our PROTEUS optimizer leads to performance close to the ones of the hand-coded execution plan.

Assuming the same physical resources, a query plan generated by the PROTEUS optimizer cannot reach the same level of performance of a hand-coded query plan because it always generates a generic query plan. This means that 1. it lacks some optimization that could be inferred only at run-time by understanding the semantic of the data and 2. adds extra operators to the query plan (e.g., for model preloading). However, the use of our Lara DSL in combination with our PROTEUS optimizer drastically reduces the amount of expertise that the end-user requires to write analytics pipelines, thus, reducing the time to knowledge.
References

