PROTEUS

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

687691

D3.10 Optimizer Prototype

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

Deliverable nature:	Demonstrator (D)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	May 31th 2018
Actual delivery date:	May 31th 2018
Version:	1.0
Total number of pages:	17
Keywords:	Optimization, Domain Specific Language, Machine Learning, Dataflow Engine, Operator Fusion

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 introduce the PROTEUS optimizer that analyzes 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 less 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.

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 first provide a comprehensive list of common optimizations for stream data processing (e.g., operator fusion, operator fission) as well as the current state-of-the-art optimizations for systems for declarative large-scale offline machine learning (i.e., SystemML and SPOOF). We start from those two types of optimizations and we build an optimizer that combines these two worlds, i.e., our optimizer performs Sum-Product optimizations by means of operator fusion.

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**	D3.9	**Title**	Optimizer Prototype
Work Package	**Number**	WP3	**Title**	Scalable Architectures for both data-at-rest and data-in-motion

Date of Delivery	**Contractual**	M30	**Actual**	M30
Status	version 1.0		final ◻	
Nature	demonstrator			
Dissemination level	public			

Authors (Partner)				
Responsible Author	**Name**	Bonaventura Del Monte	**E-mail**	bonaventura.delmonte@dfki.de
	Partner	DFKI	**Phone**	+49 30 23895 6631

Abstract (for dissemination)	
Keywords	Optimization, Domain Specific Language, Machine Learning, Dataflow Engine, Operator Fusion

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
11-Apr-2018	0.1	Bonaventura Del Monte	Initial version
29-Apr-2018	0.2	Alireza Rezaei Mahdiraji	First internal review
02-May-2018	0.3	Bonavenutura Del Monte	Addresing first internal review comments
04-May-2018	0.4	Alireza Rezaei Mahdiraji	Second internal review
07-May-2018	0.5	Bonavenutura Del Monte	Addressing second internal review comments
22-May-2018	1.0	Bonaventura Del Monte	Addressing comments from Waqas Jamil (BU) and Javier De Matias Bejarano (TREE)

Table of Contents

List of figures and/or list of tables

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 less 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.

This document is structured as follows: we provide basic background information regarding common streaming optimizations and Sum-Product class of optimization for offline machine learning. Then, we provide the description of the first prototype of a PROTEUS optimizer in Section 3. Then, we describe how we plan to extend the PROTEUS optimizer in Section 4.

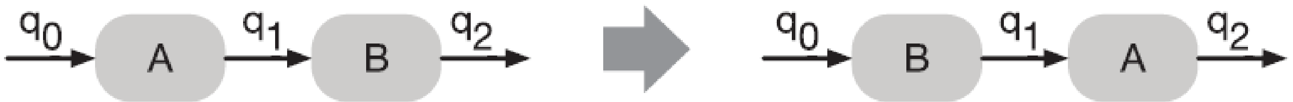
2 Background

In this section, we provide the background knowledge regarding common techniques for stream processing optimization and common optimization for execution plans meant for machine learning workloads.

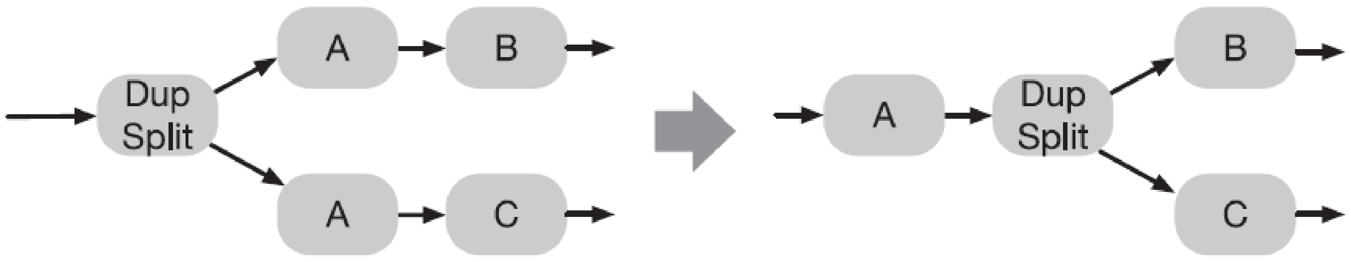
2.1 Common Streaming Optimizations

Hirzel et al. present eleven techniques for stream processing optimization [1], each with different characteristic and profitability.

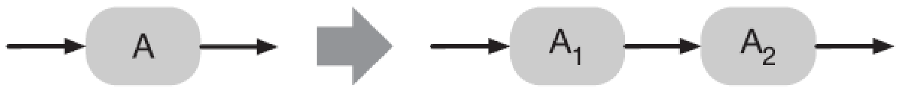
OPERATOR REORDERING (A.K.A. HOISTING, SINKING, ROTATION, PUSH-DOWN)
Move more selective operators upstream to filter data early. The key idea of operator reordering is to move operators that perform selection or projection before costly operators (e.g., join).



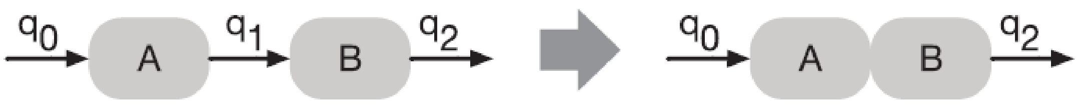
REDUNDANCY ELIMINATION (A.K.A. SUBGRAPH SHARING, MULTI-QUERY OPTIMIZATION)
Eliminate redundant computations. This optimization removes those components of the query graph that are performing redundant work. The idea is to reuse intermediate results as much as possible.



OPERATOR SEPARATION (A.K.A. DECOUPLED SOFTWARE PIPELINING)
Separate operators into smaller computational steps. The main idea behind operator separation is to enable other optimizations such as operator reordering or fission and to build parallel pipelines that scale better on multi-core systems.

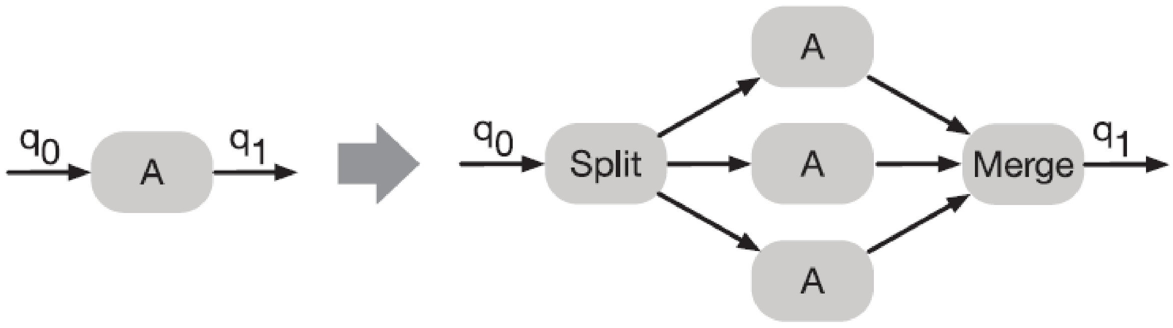


FUSION (A.K.A. SUPERBOX SCHEDULING)
Avoid the overhead of data serialization and transport. Fusion trades communication cost against pipeline parallelism. The communication cost between two fused operators is cheap. However, fused operators might run on different cores and thus having the upstream operator reading a new item, while the downstream operator is still processing the previous item.



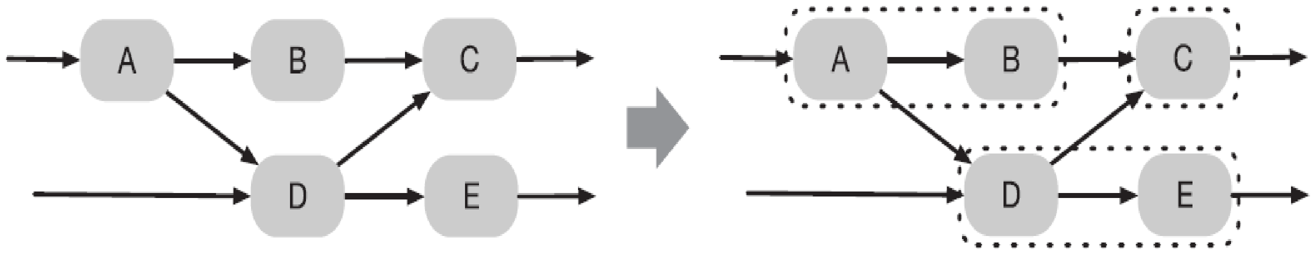
FISSION (A.K.A. PARTITIONING, DATA PARALLELISM, REPLICATION)

Parallelize computations. The intuition behind this optimization is to assign items of the input stream to different instances of the upstream operator. This requires upfront partitioning but allows for parallel execution of the upstream operator.



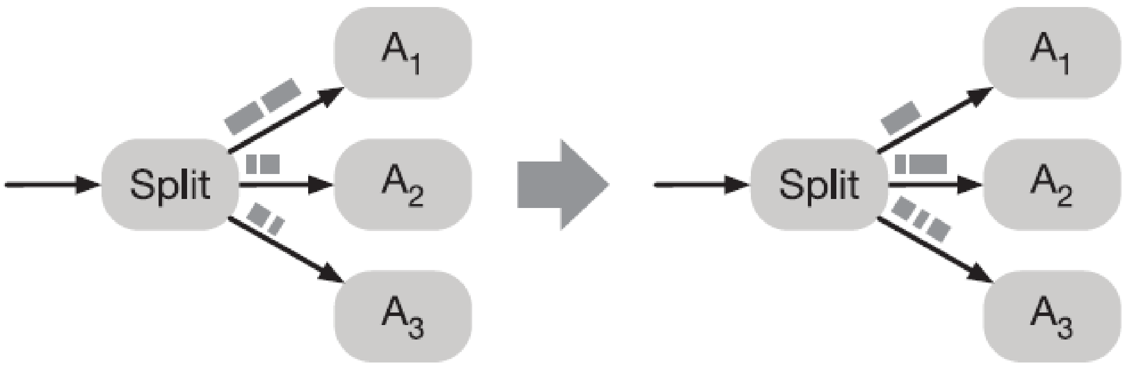
PLACEMENT (A.K.A. LAYOUT)

Assign operators to hosts and cores. The key idea behind placement is to assign specific operators to specific computing resources. For instance, if an operator requires a large amount of memory to perform its task, it would be beneficial to place it on a node equipped with more memory.



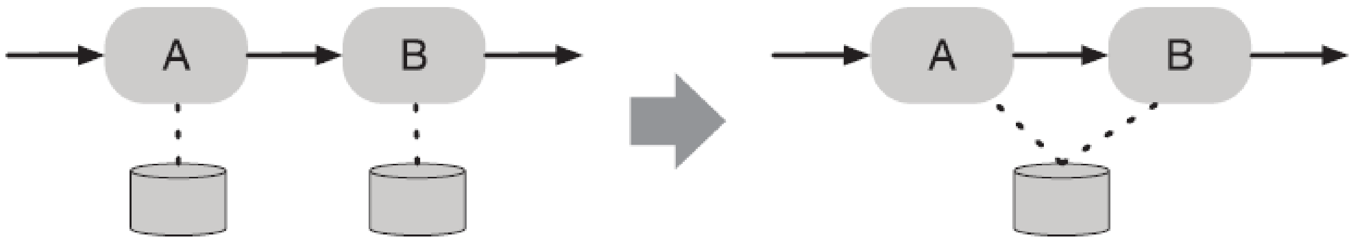
LOAD BALANCING

Distribute workload evenly across resources. The purpose of this optimization is to ensure a balanced workload on every parallel instance of an operator, thus fighting skewness.

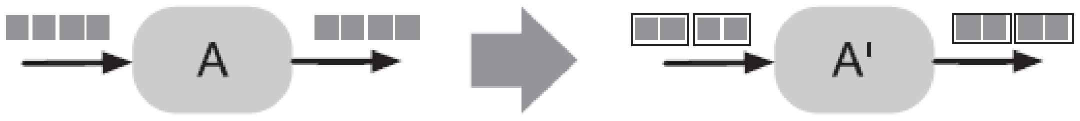


STATE SHARING (A.K.A. SYNOPSIS SHARING, DOUBLE-BUFFERING)

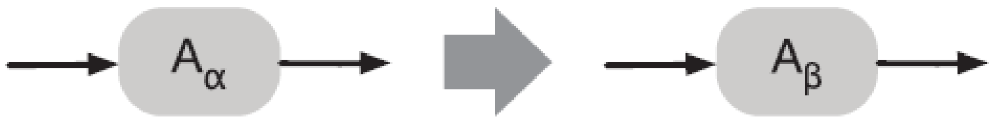
Optimize for space by avoiding unnecessary copies of data. State sharing can result in high-throughput processing as it decreases the memory footprint, which leads to less cache misses or disk I/O.



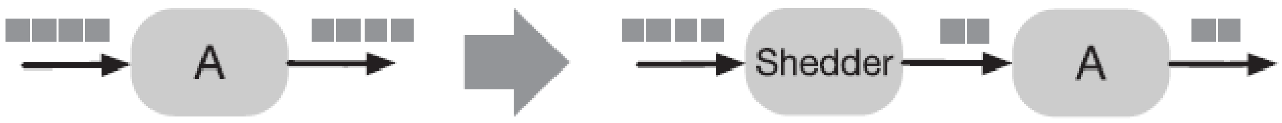
BATCHING (A.K.A. TRAIN SCHEDULING, EXECUTION SCALING)
Process multiple data items in a single batch. Batching trades latency with throughput. Batching can improve throughput by amortizing operator-firing and communication costs over more data items. Such



ALGORITHM SELECTION (A.K.A. TRANSLATION TO PHYSICAL QUERY PLAN)
Use a faster algorithm for implementing an operator.



LOAD SHEDDING (A.K.A. ADMISSION CONTROL, GRACEFUL DEGRADATION)
Degrade gracefully when overloaded. The intuition behind load shedding is to drop some input items when the operator is overloaded. Another technique that allows for load shedding is approximation.



As of May 2018, the PROTEUS platform, which is based on Apache Flink 1.4 release, supports all the above optimizations expect for the algorithm selection and the load shedding.

2.2 Machine Learning Optimizations

Several researchers proposed a number of optimizations for large-scale machine learning systems running in a batch fashion. Those systems allow the user to define machine learning algorithms through linear algebra and relational algebra and they generate efficient execution plans [2]. The main intuition behind the optimizers of those system deals with reducing the number of materialized intermediate results, reducing the number of scans of the input, and leveraging sparsity when operator chaining is involved.

Boehm et al. proposed above optimizations in their SystemML – SPO (Sum-Product Optimizer) frameworks [3]. In particular, they focused on Sum-Product Optimizations, Operator Fusion, and Sparsity exploitation [2, 4]. Briefly, SPO analyzes the execution plan produced by SystemML, rewrites the plan by fusing sum and matrix operations, and then compiles the plan through SystemML to the underlying execution engine (e.g., Apache Spark).

3 The PROTEUS Optimizer

Optimizations such as operator fusion are well-known techniques in the area of stream data processing. Stream processing engines such as Apache Flink already feature this type of optimizations. Systems designed for large scale machine learning also feature those optimizations but only in a batch execution model. In this deliverable, we develop a prototype of an optimizer that combines pure-streaming operator fusion techniques with Sum-Product optimizations. We tailor Sum-Product optimization because they can easily be applied to a variety of linear machine learning models including in SOLMA library of the PROTEUS project. We build our PROTEUS optimizer on top of the Apache Flink Streaming APIs.

Before we delve into the technical details of our solution, we present in the following a motivational example regarding how a General Linear Model such as Lasso can be mapped to dataflow execution plan. We target this specific algorithm for two main reasons: 1. it features a Sum-Product operation and 2. it is the main algorithm used in the PROTEUS validation scenario. Then, we highlight how this basic mapping results in a loss of performance and what an optimizer should do to prevent such performance degradation.

Consider the traditional formulation of the objective function of Lasso (a Linear Model):

$$\min_{w} \frac{1}{2n_{samples}}||Xw - y||_2^2 + \alpha||w||_1$$

We train Lasso using the Stochastic Gradient Descent as online optimization method and L_1 regularization. To implement that formulation, we could use a high-level language such as Scala and we would obtain something similar to the following snippet:

```scala
def computeGradient(X: Matrix, Y: Vector, oldGradient: Matrix): (Matrix, Double) = {
    val diff = X %*% W - Y // calculate activations
    val loss = diff * diff / 2.0 // calculate loss
    val newGradient = diff * data + oldGradient // calculate new gradient
    (newGradient, loss)
}

def L1Updater(grad: Matrix, oldW: Matrix, regP: Double, stepSize: Double, epoch: Int): Matrix = {
    val stepSizeCurrentEpoch = stepSize / math.sqrt(epoch)
    val W = stepSizeCurrentEpoch * grad + oldW // calculate new weights given gradient and step
    val shrinkageVal = regP * stepSizeCurrentEpoch
    W foreach {
        value => signum(value) * max(0.0, abs(value) - shrinkageVal) // apply L1 regularization
    }
}
```

The above code however is meant for single-thread execution, i.e., it is not scalable. To make it scalable, we need to reformulate it using the APIs of a dataflow engine such as Apache Flink or Apache Spark. To this end, we show in the following how a system expert would code the same snippet using the streaming APIs of Apache Flink.

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. The main logic of the Lasso algorithm is embedded in a set of User Defined Functions as follows:

1. asynchronously compute the gradient upon the arrival of a minibatch of points using the latest model available
2. asynchronously merge the gradients produced by the different workers
3. asynchronously update the current model with a merged gradient and broadcast the new model to every worker.

```
val env = ...
val trainingStream = env.addSource(...).map(Left(_))
val initialWeights = env.addSource(WeightRandomInitializer()).map(Right(_))
def stepFunc(workerIn: ConnectedStreams[Matrix, Matrix]):
(DataStream[Matrix], DataStream[Either[Matrix, Matrix]]) = {
    val worker = workerIn.flatMap(new RichCoFlatMapFunction[Matrix, Matrix, Either[Matrix, Matrix]] {
     @transient var stagedBatches = _
     @transient var currentModel = _
     override def open(parameters: Configuration): Unit = {
       stagedBatches = new mutable.Queue[Matrix]() currentModel = None
     } // incoming answer from PS
     override def flatMap2(newModel: Matrix, out: Collector[Either[Either[(Matrix, Double), Matrix], Matrix]]): Unit = { currentModel = Some(newModel)  } // incoming data
    override def flatMap1(dataOrInitialModel: Either[Matrix, Matrix], out: Collector[Either[(Matrix, Double), Matrix]]): Unit = {
    dataOrInitialModel match {
        case Left(data) => {
       if (data.isLabelled) {
          currentModel match {
             case Some(model) => {
               val diff = miniBatch % * % model - Y
               val loss = diff * diff / 2.0
               val newGradient = diff * data + oldGradient
               out.collect(Left(Left(newGradient, loss)))
             }
             case _ => stagedBatches.add(miniBatch)
          }
       } else {
          // do prediction here wrapping it with Right
       }
     }
     case Right(model) => {
      currentModel = Some(model) out.collect(Left(Right(model)))
     }
   }
  }
 }).setParallelism(workerParallelism)
 val wOut = worker.flatMap(x => x match {
   case Right(out) => Some(out)
   case _ => None
 }).setParallelism(workerParallelism)
 val ps = worker.flatMap(x => x match {
   case Left(workerOut) => Some(workerOut)
   case _ => None
 })
 .setParallelism(workerParallelism)
 .partitionCustom(new Partitioner[Int]() {
   override def partition(key: Int, numPartitions: Int): Int = key % numPartitions
 }, paramPartitioner).flatMap(
    new RichFlatMapFunction[Either[(Matrix, Double), Matrix], Either[Matrix, Matrix]] {@
      transient val currentModel = _ override
      def open(p: Configuration): Unit = {
        super.open(p)
        currentModel = None
      }
      override def flatMap(msg: Either[(Matrix, Double), Matrix],
           out: Collector[Either[Matrix, Matrix]]): Unit = {
        msg match {
          case Left(t) => {
            val tmp = currentModel.get * t._1
            val stepSizeCurrentEpoch = stepSize
            val W = stepSizeCurrentEpoch * grad + (tmp / 2)
            val shrinkageVal = regP * stepSizeCurrentEpoch
            W = W foreach {
             value => signum(value) * max(0.0, abs(value) - shrinkageVal)
            }
            out.collect(W)
          }
          case Right(model) => currentModel = Some(model)
        }
      }
    }).setParallelism(psParallelism)
  val psToWorker = ps.flatMap(_ match {
    case Left(x) => Some(x)
    case _ => None
  }).setParallelism(psParallelism) // TODO avoid this empty map?
  .map(x => x)
  .setParallelism(workerParallelism)
  .partitionCustom(new Partitioner[Int]() {
    override def partition(key: Int, numPartitions: Int): Int = {
      if (0 <= key && key < numPartitions) {
        key
      } else {
        throw new RuntimeException("Pull answer key should be the partition ID itself!")
      }
    }
 }, wInPartition)

 val psToOut = ps.flatMap(_ match {
   case Right(x) => Some(x)
   case _ => None
 })
.setParallelism(psParallelism)
val wOutEither: DataStream[Either[Either[(Matrix, Double), Matrix], Matrix]] =
    wOut.forward.map(x => Left(x))
val psOutEither: DataStream[Either[Matrix, Matrix]] =
  psToOut.forward.map(x => Right(x))(
  psToWorker,
  wOutEither
   .setParallelism(workerParallelism)
   .union(psOutEither.setParallelism(psParallelism))
 )
}

trainingStream
       .countWindow(miniBatchSize)
       .apply(toMatrixTransformer)
       .union(initialWeights)
       .iterate(x: ConnectedStream[Matrix, Matrix] => stepFunc(x), iterationWaitTime)

```

3.1 Compilers and Optimizers to the Rescue

The example in the above section shows how a simple algorithm such as Lasso result in a muddle of system-specific constructs as well as language-specific aspects. However, the user really expects to have something like:

```
val  env  =  ...
val  trainingStream  =  env.addSource(...).toMatrix
val  initialModel  =  env.addSource(LassoModelRandomInitializer()).toMatrix
withFeedback(trainingStream,  initialModel)  {
    (X, model)  =>  {
        val d = diag(DenseMatrix(X.colsNum)(sqrt(abs(model.gamma))))
        val  A  =  model.A + X %*% X.t + inv(d)
        val  prediction  =  model.b.t %*% inv(A) %*% X
        (prediction, (A, gamma))
    },
    (currModelA, currModelB) => currModelA + currModelB,
    (currModel, Y, X)  =>  {
        val l = Y %*% X
        currModel.b += l
    }
}
```

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 D4.3), 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

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 matrixes 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 matrixes. Consider the following formula, which is a common subexpression present in any General Linear Model:

$$x * W - y$$

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 matrixes 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 matrixes 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.

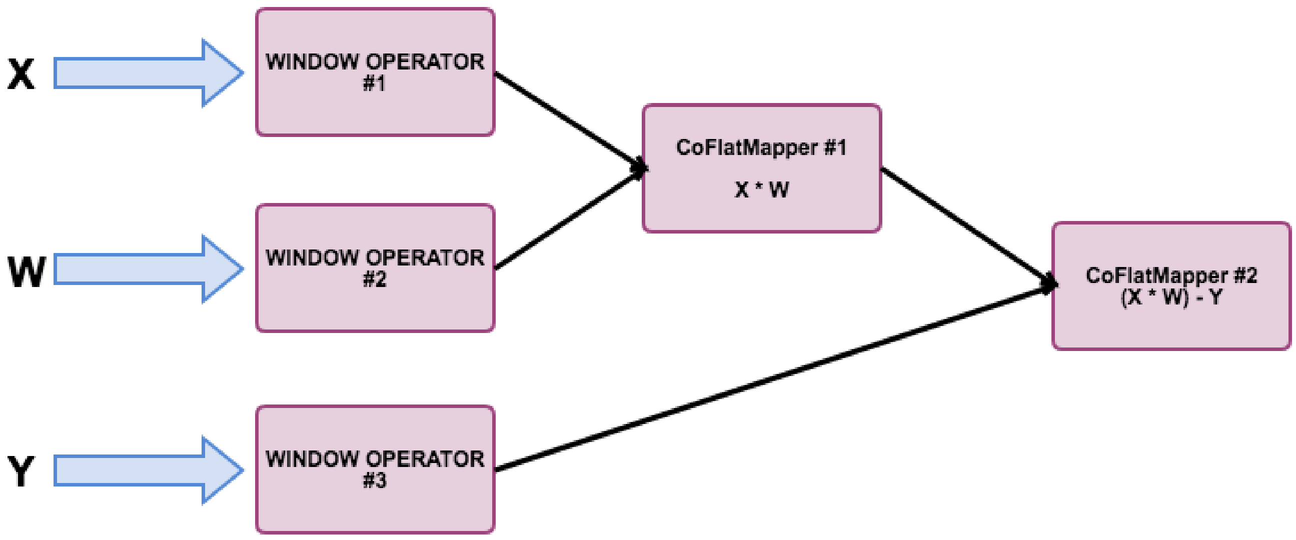


Figure 1 Simple execution plan for X * W – Y.

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.

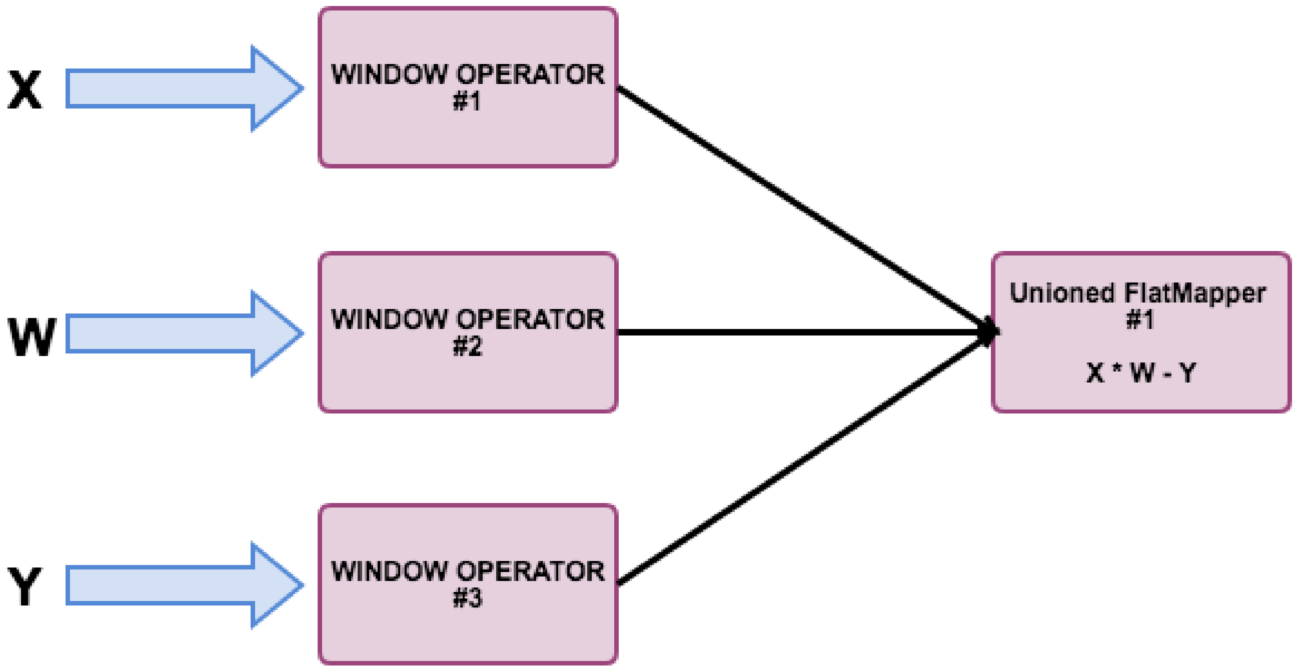


Figure 2 Optimized execution plan for X * W – Y.

4 Conclusions

In this document, we presented a first prototype of the PROTEUS optimizer. The main feature of this optimizer is to holistically optimize a logical execution plan containing linear algebra operators performing matrix operations on streams. The proposed optimizer is able to perform Product Sum optimizations by means of operator fusing. This leads to better resource utilization as the underlying engine will have to schedule less physical operators.

As future work, which we will carry out in D3.11, we plan to benchmark the capabilities of the PROTEUS optimizer by using it on one of the algorithm provided by WP4.

The code of the optimizer is available on the PROTEUS Github page at the following address: https://github.com/proteus-h2020/streaming-lara

References

[1] A Catalog of Stream Processing Optimizations, Hirzel et al., ACM Computing Surveys, 2014

[2] On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML, Hirzel et al., *CoRR,abs/1801.00829*, 2018

[3] SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs, Boehm et al., *IEEE Data Eng. Bull. 37*(*3*), 2014

[4] SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning, Elgamal et. al, CIDR, 2017