PROTEUS

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

687691

D3.9 Declarative Language Tested Implementation

Lead Author: Jeyhun Karimov
With contributions from:
Bonaventura Del Monte, Alireza Rezaei Mahdiraji

Reviewer: Waqas Jamil (BU), Hamid Bouchachia (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:	
Keywords:	Hybrid processing, online machine learning, Lasso, performance evaluation, benchmark

Abstract

In PROTEUS, we propose a declarative language to ensure non-imperative style of programming in which a user describes the desired workflow without systems programming and explicitly listing low-level commands. In D3.7 and D3.8, we describe the main ideas behind our solution called Lara. The goal of this deliverable is to test our solution. In the previous deliverables, we tested our solution only with synthetic and not the PROTEUS use-case data. Furthermore, we utilize online Lasso algorithm which is developed in SOLMA library in WP4 of PROTEUS. Moreover, we highlight the hybrid engine capabilities using PROTEUS use-case and online Lasso algorithm. In this deliverable, we address the limitations above by conducting a benchmark with actual PROTEUS use-case, online Lasso. Our benchmark suite features a real data set derived from ArcelorMittal use-case of PROTEUS. Moreover, we incorporate the hybrid capabilities of PROTEUS engine in our use-case and conduct benchmarks accordingly.

Executive summary

In this deliverable, we benchmark the declarative language proposed in D3.8. As a use-case for our benchmark, we chose the online Lasso (Least Absolute Shrinkage and Selection Operator) algorithm. The details of online Lasso, which is an algorithm based on regression model, has been described in the deliverable D4.3 of PROTEUS and the original algorithm can be found in [1].

The first issue that we address in our benchmark is usage of the PROTEUS use-case dataset. In previous deliverables, we adopted mostly synthetically generated data or publicly available data sets. Although this is acceptable to for some use-cases we should use the dataset provided by ArcelorMittal as the industrial partner of the PROTEUS project in our evaluations and in the end product.

The second issue we address in our benchmark is performance evaluation of hybrid engine capabilities. Bridging the gap between streaming and batch processing was one of the main intuitions behind PROTEUS. To overcome this challenge, we proposed side inputs, developed on top of Apache Flink [2]. In this deliverable we also benchmark the hybrid capability of our framework with online Lasso use-case.

The third issue we address is a high-level implementation of online Lasso using the distributed streaming LARA language features which we have introduced in the deliverable D3.8. In streaming LARA data model, an input stream is a growing matrix where each new item of the stream is a row of the streaming matrix. Streaming LARA updates existing linear algebra models upon receiving each new streaming element by performing operations on part of streaming matrixes (see the deliverable D3.8 for more details). In this deliverable we define metrics for our benchmark suite and conduct benchmark with online Lasso.

We elaborate the existing benchmark models and analyze which model is eligible for our benchmark: open, closed and partly open models. We discuss which model is eligible for our benchmark scenario. We also describe the data types used in our benchmark, i.e., PROTEUS dataset with its unique features.

To benchmark our solution, we need to provide an input dataset for a system under test. In our test scenario, we integrate the PROTEUS hybrid operator with the online Lasso algorithm as well. The main intuition is that we use hybrid operator for training the online Lasso model before online predictions to increase the algorithm accuracy. In our main input source, provided by ArcelorMittal, consists of one file. The main issue is to separate the overall input into two pieces, one part for the hybrid operator and training the initial online Lasso model and one part for the online update of the initial Lasso model, is non-trivial. In the main input file of ArcelorMittal data set, the coil and flatness measurements are not ordered by timestamp ~~Because this file is real and industry driven dataset, this behaviour is acceptable~~ because, sensors can send out-of-order data and as a result, overall input data set might be completely unordered.

We configure our pub/sub system to provide max performance and conduct our experiments to measure latency. We consider two main types of latency: average end-to-end latency and hybrid operator latency. These metrics are specific for this deliverable as we benchmark only a component of PROTEUS. We have more general KPIs for the whole PROTEUS project which are defined in other deliverables, e.g., D2.10.

We run our benchmark on random portions of data, each for one hour, this is time efficient since, performing a single experiment takes several days. We start each experiment with predefined input offset in Apache Kafka and measure our metrics.

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**	Declarative Language Tested Implementation
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**	Jeyhun Karimov	**E-mail**	Jeyhun.Karimov@dfki.de
	Partner	DFKI	**Phone**	*+49 30 23895 5359*

Abstract (for dissemination)	In PROTEUS, we propose a declarative language to ensure non-imperative style of programming in which a user describes the desired workflow without systems programming and explicitly listing low-level commands. In D3.7 and D3.8, we describe the main ideas behind our solution called LARA. The goal of this deliverable is to test our solution. In the previous deliverables, we tested our solution only with synthetic and not the PROTEUS use-case data. Furthermore, we utilize online Lasso algorithm which is developed in SOLMA library in WP4 of PROTEUS. Moreover, we highlight the hybrid engine capabilities using PROTEUS use-case and online Lasso algorithm. In this deliverable, we address the limitations above by conducting a benchmark with actual PROTEUS use-case, online Lasso. Our benchmark suite features a real data set derived from ArcelorMittal use-case of PROTEUS. Moreover, we incorporate the hybrid capabilities of PROTEUS engine in our use-case and conduct benchmarks accordingly.
Keywords	Hybrid processing, online machine learning, online Lasso, performance evaluation, benchmark

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
28.04.2018	0.1	Jeyhun Karimov	Initial version
01.05.2018	0.2	Alireza Rezaei Mahdiraji	Internal review
19.05.2018	0.3	Hamid Bouchachia and Waqas Jamil	Review from BU
23.05.2018	1.0	Jeyhun Karimov	Addressing reviews of BU

Table of Contents

List of Figures

List of Tables

Abbreviations

HDFS: Hadoop Distributed File System

LASSO: Least Absolute Shrinkage and Selection Operator

1 Introduction

Declarative programming is one of the essential building blocks in today's enterprise technology [5]. The main idea is that most of the enterprises embrace more than one aspect of technology. The declarative languages decouple the developers from low-level system programming, which allows to by-pass the problem of knowing the details of the underlying processing and their respective theory, which consequently reduces the cost of the company - less experts are employed. In the deliverable D3.8, we propose LARA declarative language which serves as the essential building block for scalable online data processing. The declarative language LARA leads to the development of Scalable Online Learning Machine Algorithms (SOLMA) library.

In D3.6 we provided the syntax of declarative language. In D3.7 we implemented the first prototype, and the completion of the prototype in D3.8. In this deliverable, our focus is on testing of the declarative language.

We consider the use-case as our benchmark and we use online Lasso (Least Absolute Shrinkage and Selection Operator) as our learning algorithm. The details of Lasso as a variation of regression model has been described in the deliverable D4.3 of PROTEUS. Although we customize Lasso for PROTEUS in D4.3, the original algorithm can be found in [1]. We utilize online Lasso algorithm for flatness prediction in real-time.

ArcelorMittal data possesses a wide range of variables, so, we benefit from variable selection and regularization features of online Lasso. Moreover, online Lasso automatically does model selection to improve the overall quality of regression, which is very important feature for PROTEUS.

The first issue we address in our benchmark is usage of the PROTEUS use-case dataset. Herein we use ArcelorMittal dataset, which has a unique set of features that are difficult to find in other publicly available datasets.

The proposed solution is not limited to a specified dataset. We present a general framework, that can accommodate variety of machine learning algorithms as there is no single machine learning algorithm that can perform well on all data sets.

The second issue we address in our benchmark is performance evaluation of hybrid engine capabilities. Bridging the gap between sequential and batch processing is one of the main objectives of PROTEUS. To overcome this challenge, we proposed side inputs in D3.2, developed on top of Apache Flink [2]. With side inputs, instead of using DataSet and DataStream APIs of Apache Flink, we only use DataStream API and treat historical data as bounded stream data. We train an offline our model using side inputs to load a historical dataset. We take the hybrid capabilities into account while defining metrics, further details can be found in Section 2.4.

The third issue we address is a high-level implementation of online Lasso using the distributed streaming LARA language features which we have introduced in the deliverable D3.8. In streaming LARA data model, an input stream is a growing matrix where each new item of the stream is a row of the streaming matrix. Streaming LARA updates existing linear algebra models upon receiving each new streaming element by performing operations on part of streaming matrixes (see the deliverable D3.8 for more details).

2 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. Fourthly, we analyze the challenges we encountered while benchmarking hybrid operators of PROTEUS engine and provide our solutions. Finally, we provide the main metrics we adopt in this benchmark and give their definition.

2.1 Benchmark Model

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

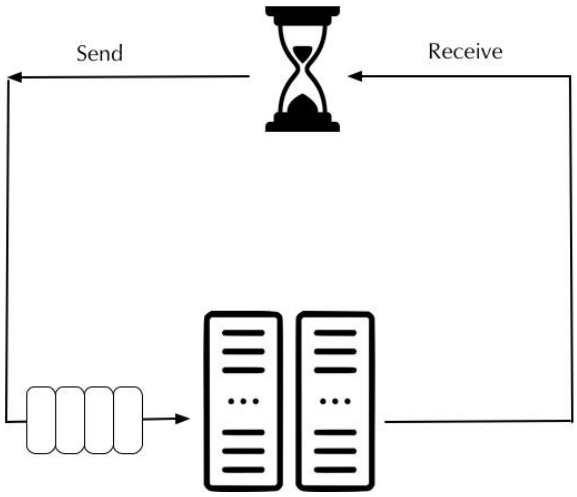

Figure 1. Closed benchmark model

The first model is called closed model. As we can see from Figure 1, in a closed model new requests for system under test is triggered once the previous is completed. The new request might also have been 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 latter 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 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 the website. Secondly, the user receives respective response from the website. The new request might be initiated immediately after some time delay. Let N^{think} be the number of users thinking for some time interval before initiating a new request, N^{run} be the number of users running queries in the system under test, and N^{system} be the number of users queued to run some requests in the system under test. Then, $N^{think} + N^{system} + N^{run} = N$.

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.

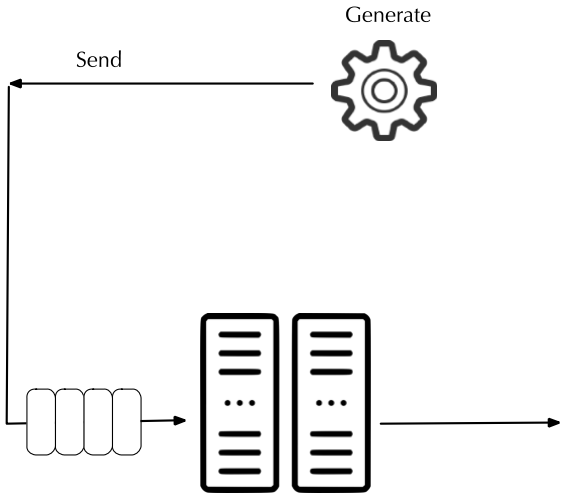

Figure 2. Open benchmark model

The second model is called open model. As we can see from Figure 2 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 asynchronous 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 fixed amount of users for 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.

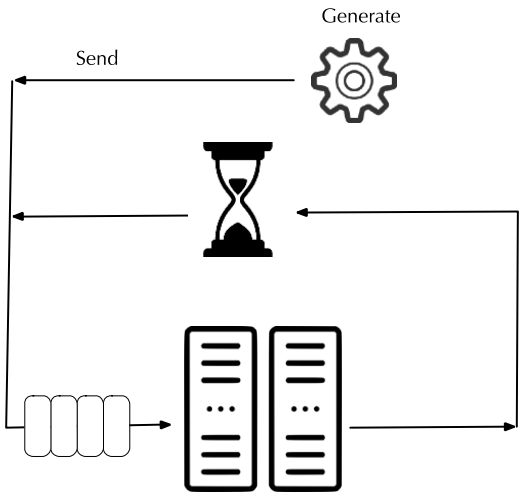

Figure 3. Partly open benchmark model

The third model is called partly open model. As we can see from Figure 3 this model lies between the open and close models. 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 online 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.

2.2 Data Description

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

coilId	x	y	slice	data

Figure 4. Tuple fields for SensorMeasurement2D

Figure 4 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

coilId	x	slice	data

Figure 5. Tuple fields for SensorMeasurement1D

Figure 5 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.

2.3 Use-case

We adopt online Lasso algorithm for our use-case in this benchmark. The detail of this algorithm is provided in D4.3. The main reason to choose online Lasso is that it is implemented in SOLMA and used as main algorithm to address the flatness prediction in the PROTEUS project. We provide delayed feedback to our online Lasso model to train existing model in a real-time fashion.

Another reason for adopting online 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 online Lasso regularization, we can easily handle sparse vectors.

Yet another reason for adopting online 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 online Lasso and without prior blocking operations.

2.4 Input Data for Hybrid Operator

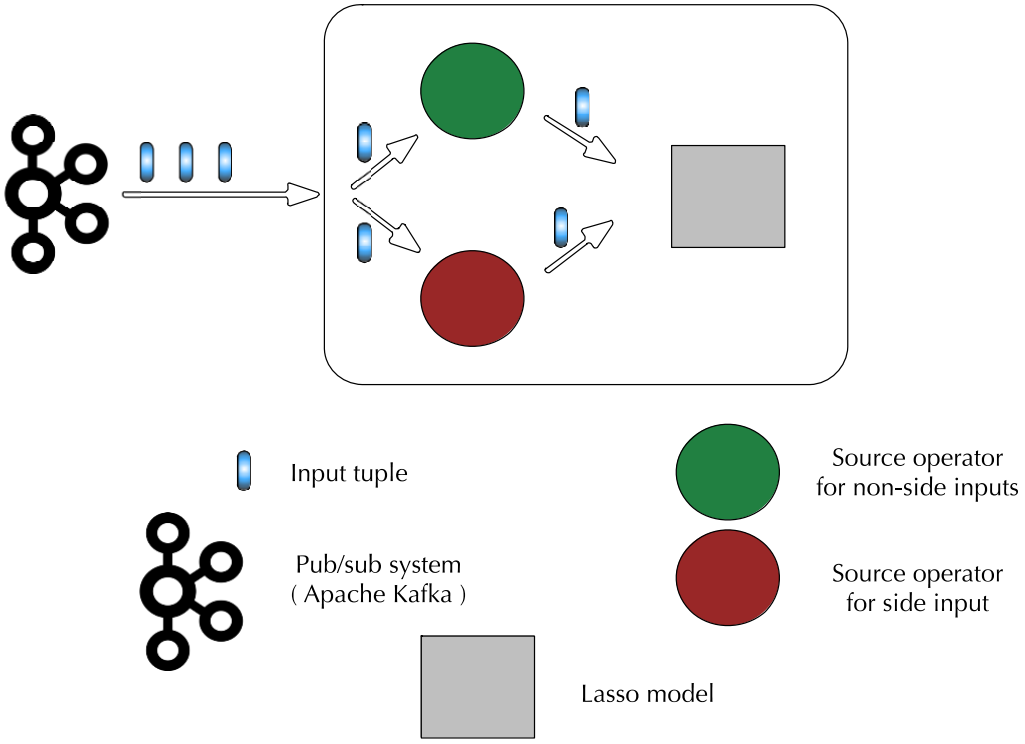

Figure 6. First scenario: Using hybrid operator in online Lasso. Both hybrid operator and prediction operators in online Lasso consume input from the main input source.

To benchmark our solution, we need to provide an input data set for a system under test. In our test scenario, we integrate the PROTEUS hybrid operator to online Lasso algorithm as well. The main intuition is that we use hybrid operator for pre-training the online Lasso model before online predictions. This increases the algorithm accuracy and performance.

The main issue is that separating the overall input into two pieces, one for the hybrid operator and one for the rest, is non-trivial. In the main input file, the coil and flatness measurements are not ordered by timestamp.

Because this file is real and industry driven data set, this behaviour is acceptable. This happens because sensors can send out-of-order data and as a result, overall input data set might be completely unordered.

To separate the input data for training the model inside hybrid operator and prediction, we considered two cases. The first case is shown in Figure 5. Basically, we read input from pub/sub system. Let the size of our input data set be S, the size of the input data for training (for hybrid operator) be S^T, and the size of input data for predicting be S^P. Obviously, $S = S^T + S^P$. The intuition behind Figure 5 is that we separate the first S^T portion of input for hybrid operator and the rest for prediction. As we mentioned above, the problem with this approach is that the input data is not sorted with respect to measurement time. So, in this case, the coils appeared in S^T portion might appear in S^P portion as well, which is not acceptable. As a result, we need to separate the input data such that there will be two data sets with disjoint coilId.

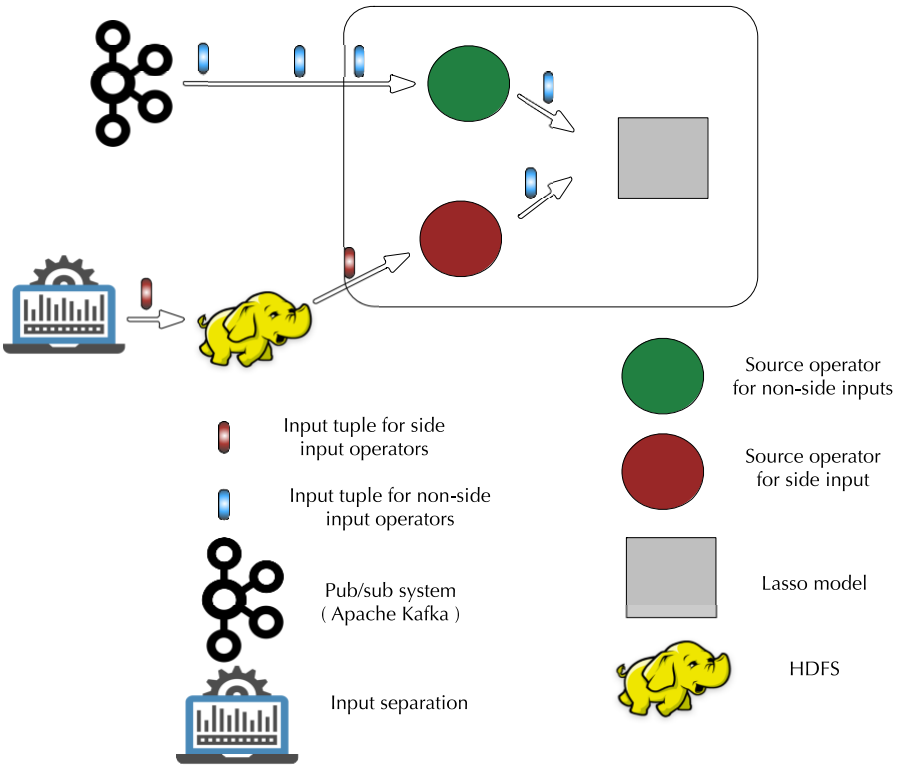

Figure 7. Second scenario: Using hybrid operator in online Lasso. Hybrid operator consumes input from HDFS, which is the output of pre-processing of main input and prediction operators in online Lasso consume input from the main input source.

To solve the problem described above, we separate the input data set into two sets. Firstly, we sort the overall data set with respect to coilId field. Secondly, we extract the first S^T portion from the sorted data set, which will be the input data set for hybrid operator. Finally, we perform left join between the original data set and the sorted data set and extract all tuples belonging to sorted data set. The overall process is shown in Figure 7.

As we can see from Figure 6, we perform our initial input separation described above and save the input for hybrid operator to HDFS and the input for prediction operators of online Lasso to sub/sub system. Then, hybrid operator reads its input from HDFS and performs training on data for online Lasso model and prediction operators of Lasso read from pub/sub system and perform predictions.

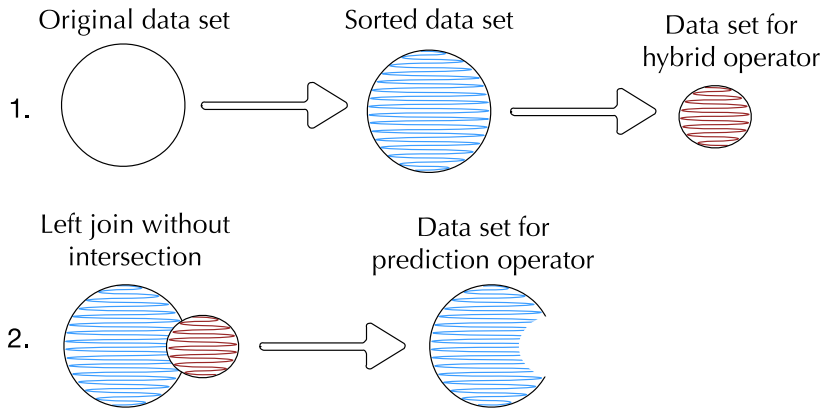

Figure 8. Overall intuition behind data set separation for hybrid and prediction operators

2.5 Metrics

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 pub/sub system to provide max throughput and conduct our experiments to measure latency. We consider two main types of latency: average end-to-end latency and hybrid operator latency.

2.5.1 End-to-End Latency

Our first 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 online 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:

$$L_{avg} = T(S) / S \quad (1)$$

2.5.2 Hybrid Operator Latency

Our second latency definition is hybrid operator latency. We define hybrid operator latency as the time needed to acquire input for hybrid operator and process it divided by the overall number of tuples in S^H as shown in Equation 2:

$$L_{hyb} = T(S^H) / S^H \quad (2)$$

3 Experimental Evaluation

We conduct our experiments on Intel Xeon CPU E5530 2.40GHz with 16 CPUs. We use 2- and 4-node cluster configurations. Because PROTEUS framework simulates the input data set, the overall runtime takes several days. The reason is that the input data set is derived from real production environment running for several days.

We run random portions of data for our benchmark, each for one hour, as repeating all the experiments takes several days, which is not feasible. For each experiment, we start the experiment with predefined input offset in Apache Kafka and measure our metrics.

Figure 9 shows benchmark strategy used in this deliverable. In the upper side of the figure there is a queue. This is a pub/sub system, specifically Apache Kafka. Each tuple has its own timestamp and we adopt *event-time processing* semantics. We show the experiments with E1, E2, and E3. Basically, experiments start and end with different offsets in Apache Kafka. Given the start and end points of an experiment, we submit this information to cluster for execution. We have several small clusters within the large cluster. We denote clusters with blue and black rounded rectangles in the figure. The clusters are used for parallel execution of different experiments. Finally, we accumulate the results of our experiments in the end.

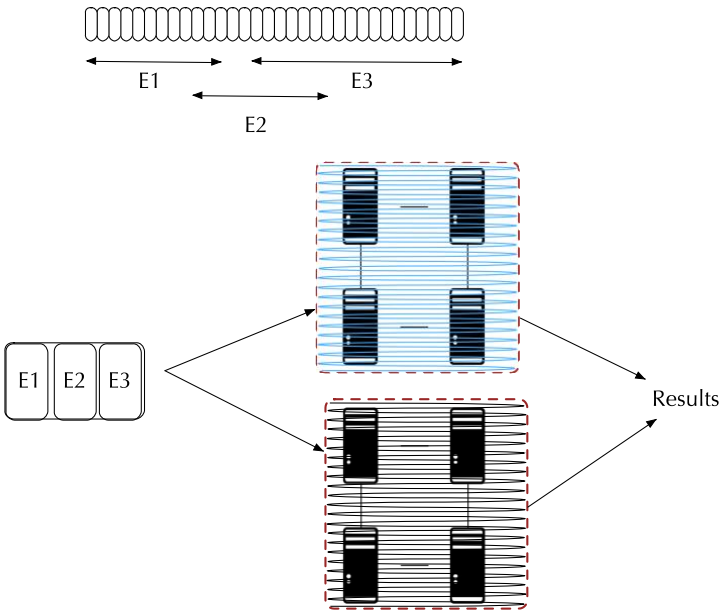

Figure 9. Benchmark strategy

	2-node	**4-node**
Average Latency	0.64	0.51
Min Latency	0.54	0.43
Max Latency	1.12	1.31
Percentiles (90, 95, 99)	(0.91, 0.93, 1.06)	(0.91, 0.98, 1.04)

Table 1. End-to-end latency measurements (in seconds) with 40% training data

Table 1 shows end-to-end latency. We run experiment for an hour and provide training data for hybrid operator input size as 40% of overall input. 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 warm-up in the beginning of the experiment.

	2-node	**4-node**
Average Latency	0.69	0.54
Min Latency	0.54	0.46
Max Latency	1.20	1.33
Percentiles (90, 95, 99)	(0.91, 0.95, 1.09)	(0.92, 0.98, 1.05)

Table 2. End-to-end latency measurements (in seconds) with 20% training data.

Table 2 shows end-to-end latency run for an hour with training data for hybrid operator input size as 20% of overall input. When compared with Table 4, we can see the latency is higher with 40% training data. The reason is that amount of training data impact the overall latency. That is, with less amount of training data the overall latency is high because the overall number of input tuples is less and any side effect, such that low system start-up, cold memory, and etc, influences more the overall latency and results in higher latency.

Table 3 shows the latency measurements specifically for hybrid operator with 40% training data of our overall experimental input. Although the training process is costlier than prediction part, when we take average latency for each tuple, the difference between this and end-to-end latency is small and can even be neglected for some use-cases.

	2-node	**4-node**
Average Latency	0.79	0.60
Min Latency	0.66	0.54
Max Latency	1.20	1.41
Percentiles (90, 95, 99)	(0.96, 0.98, 1.09)	(0.97, 0.99, 1.08)

Table 3. Hybrid operator latency measurements, 40% training data.

Table 4 shows the latency measurements specifically for hybrid operator with 20% training data of our overall experimental input. As expected the overall measurements are lower than the ones with 40% training data because the amount of processed data is doubled, as a result the impact of side effects in overall result is decreased. However, the latency measurements increase around 3% when we double the amount of training data. Considering this fact and the results when we scale-out, we can conclude that our system can handle online machine learning, in particular online Lasso, in scale.

	2-node	**4-node**
Average Latency	0.81	0.67
Min Latency	0.67	0.57
Max Latency	1.24	1.5
Percentiles (90, 95, 99)	(1.01, 1.04, 1.09)	(1.02, 1.09, 1.15)

Table 4. Hybrid operator latency measurements, 20% training data.

As we can see, end-to-end latency is on average less than hybrid operator latency. The reason derives from the definition of end-to-end latency. Basically, end-to-end latency includes hybrid operator latency for training time and prediction operator latency after training. In our experiments, we chose the input size for prediction operator more than the one for hybrid operator. As a result, the impact of prediction operator on end-to-end latency is more than hybrid operator because the number of input tuples for prediction operator is more and prediction operator performs with less latency than hybrid operator.

4 Conclusion

Declarative language is one of the important building blocks of PROTEUS. We provided our declarative language implementation and the theory behind it in deliverables D3.6, D3.7, and D3.8. In this deliverable we utilize the findings from the previous deliverables and provide tested implementation of declarative language. Our findings are twofold. Firstly, we describe overall benchmark strategy and use-case for our benchmark. We provide necessary metrics used in our benchmark. Secondly, we conduct our experiments with real PROTEUS data provided by our use-case partner, ArcelorMittal. In previous deliverables, we mostly adopted synthetically generated data or publicly available data sets. However, it is important to see the measurements of our metrics with real PROTEUS data. Thirdly, we also test the hybrid operator, which is an important part of PROTEUS engine. As a result, we conduct that the overall measurement results are in the order of millisecond on average and our findings show that the measurement results get better as we scale our framework.

References

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

[2] Carbone, Paris, et al. "Apache Flink: Stream and batch processing in a single engine." *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015).

[3] Schroeder, Bianca, Adam Wierman, and Mor Harchol-Balter. "Open Versus Closed: A Cautionary Tale." *Nsdi*. Vol. 6. 2006.

[4] https://github.com/scalanlp/breeze

[5] Lloyd, John W. "Practical Advantages of Declarative Programming." *GULP-PRODE (1)*. 1994.