PROTEUS

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

687691

D3.2: Hybrid computation prototype implementation

Authors: Bonaventura Del Monte (DFKI)

With contributions from: Asterios Katsifodimos

Reviewers: Alvaro Agea, Daniel Higuero (LMBDP)

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	31/11/2017
Actual delivery date:	30/11/2017
Version:	1.01
Total number of pages:	23
Keywords:	Hybrid computing architecture prototype

Abstract

The standard approach for analysing high-volume data streams (e.g., log data) is to run periodic batch jobs (e.g., hourly). Batch processing frameworks (e.g., Hadoop) primarily focus on job throughput and often have difficult handling latency- sensitive jobs, e.g., from interactive analyses. In PROTEUS there is need for low latency responses to potentially complex queries over high-volume, infinite streams of data: this demands online processing capabilities with access to historical data.

In this deliverable, we built a first prototype of a computing model able to handle fast streams and large quantities of batch data. A novel operator supporting blocking side inputs has been introduced in order to mix streamed data with multiple batch stored datasets.

Executive summary

This deliverable deals with the creation of a first prototype of a data analytics and workflow platform meant to analyse streaming and historical data coming from Arcelor Mittal's (AMIII) sensors installed in their factory infrastructure. We called the target platform a *hybrid processing engine*, for its ability to deal with both streaming and historical data under the same programming model. The hybrid processing engine will enable PROTEUS partners to develop analysis algorithms and predictive models which will take into account both the historical data that AMIII has been collecting since more than a decade, as well as the fresh, and fast data which come from the sensors in the manufacturing plants in real time. This hybrid processing engine will form the basis for the development of PROTEUS.

The standard approach for analysing high-volume data-streams (e.g., log data, or sensor data) is to run periodic *batch* jobs (e.g., hourly). Batch processing frameworks like Hadoop primarily focus on job throughput and often have issues handling latency-sensitive jobs, e.g., from interactive analyses. There is often a need for low latency responses to potentially complex queries (e.g., statistical analysis, machine learning) over high-volume, infinite streams of data: this demands online processing capabilities. Note that batch processing provides rigorous results because it can use more data and, for instance, perform better training of predictive models. The use case of AMIII, is a good example of a time-critical application which required deep analysis of data: the data coming from sensors must be examined and checked against historical data. As soon as there is any anomaly detected (e.g., malformed steel coils) the system has to alert the factory workers in order to fix the issues. Thus, here we witness the need for an integrated platform which is able to provide instant answers and detect anomalies (low latency) but also to be able to use historical data in order to e.g., train predictive models, or even perform joins of historical data against low-latency streams of sensor reads.

In the following pages, we will discuss about our first implementation of our hybrid processing engine: first of all, we will analyse the overhauled execution engine able to mix streaming data with multiple historical dataset. Then, we will show the current user-level programming API. Moreover, in order to better assess the abilities of the new engine, we will show a basic example which will deal with the training of a simple linear regression model on streaming and batch data.

The result of this deliverable is available online as stated in section 1.2.

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**	D3.2	**Title**	Hybrid computation prototype implementation
Work Package	**Number**	WP3	**Title**	Scalable Architectures for both data-at-rest and data-in-motion

Date of Delivery	**Contractual**	M12	**Actual**	M12
Status	version 1.01		final	
Nature	report			
Dissemination level	public			

Authors (Partner)	Bonaventura Del Monte (DFKI)			
Responsible Author	**Name**	Asterios Katsifodimos	**E-mail**	asterios.katsifodimos@tu-berlin.de
	Partner	DFKI	**Phone**	0171 549 5731

Abstract (for dissemination)	The standard approach for analysing high-volume data streams (e.g., log data) is to run periodic batch jobs (e.g., hourly). Batch processing frameworks (e.g., Hadoop) primarily focus on job throughput and often have difficult handling latency- sensitive jobs, e.g., from interactive analyses. In PROTEUS there is need for low latency responses to potentially complex queries over high-volume, infinite streams of data: this demands online processing capabilities with access to historical data. In this deliverable, we built a first prototype of an efficient computing model able to handle fast streams and large quantities of batch data. A novel operator supporting blocking side inputs has been introduced in order to mix streamed data with multiple batch stored datasets.
Keywords	Hybrid computing architecture prototype

Version Log			
Issue Date	**Rev. No.**	**Author**	**Change**
21/11/2017	1.0	Bonaventura Del Monte	Changes based on reviews of Alvaro Agea and Daniel Higuero
23/11/2017	1.01	Bonaventura Del Monte	Changes based on reviews of Marcos Sacristán Cepeda

Table of Contents

List of figures and list of tables

List of listings

Abbreviations

Acronym	Definition
ML	Machine Learning
LR	Linear Regression
UDF	User Defined Function

1 Introduction

The standard approach for analysing high-volume data streams (e.g., log data) is to run periodic batch jobs (e.g., hourly). Batch processing frameworks (e.g., Hadoop [1]) primarily focus on job throughput and often have difficult handling latency- sensitive jobs, e.g., from interactive analyses. There is often a need for low latency responses to potentially complex queries over high-volume, infinite streams of data: this demands online processing capabilities. Although data management on streams is not new, how to best integrate batch and online processing ensuring simplicity, maintainability, efficiency still remains an open question. This is the challenge the prototype presented in this document tries to overcome.

Hybrid (i.e., streaming and batch) data processing has become increasingly important in modern applications. We define the hybrid system as a combination of stream and batch processing in real time. There are many applications of the hybrid system from machine learning to anomaly detection. To implement such a system, there are requirements to be considered. In PROTEUS we are dealing with sensor data and one goal is efficiently process it and do operations between historical data to detect anomalies. Table 1 shows the general requirements for the hybrid system. We discuss in more detail in the following sections.

Low latency	In the context of PROTEUS, we process the data coming from sensors. Processing this data with low latency is essential to detect the defects in real-time. Currently, detection of defect can take up to several days which is unacceptable in most cases and can cause unforeseen results in the production.
High availability	If the sensor data is massive, we need to provide distributed system to process the data and therefore, availability of such system becomes essential. For example, if any computation or data processing unit goes down for any reason, the system should continue to operate and detect the defects.
High level APIs	Another requirement for the hybrid system is flexible data processing system that can be configured with high level APIs. There is a probability that the company's production system and computational data processing logic can change in future. The requirement indicates that the amount of code changes in such scenario should be minimal.
Scalability	Especially in the scope of this project, the amount data that obtained from sensors can be massive. Therefore, the hybrid system should handle this and the computational time should be competitively same (and not exponentially increasing) assuming that we increase count of computational units.

Table 1. General requirements for hybrid processing system.

1.1 Document structure

We implemented the aforesaid hybrid computational model in Apache Flink [2]. Therefore, we first, introduce Flink in Section 2. Moreover, we analyse the differences between DataSet and DataStream APIs of Flink in order to give readers a background needed for better understanding our new API and execution environment. In section 3, we describe our changes in Flink's execution engine to support our aforementioned computational model and operator design. In section 4, we show an example of execution of our prototype, dealing with a linear regression model training. Finally, we conclude by giving the main take-home messages of this document.

1.2 Deliverable result

The result of this demonstrator deliverable is available as a git[1] open-source repository at the following link [3]. To deploy our prototype framework, one must first build it from sources using maven[2].

The source can be obtained cloning the git repository with the following command:

```
git clone –b proteus https://github.com/proteus-h2020/proteus-engine.git
```

Since our system is based on Flink, the very same building instructions applies [4].

[1] https://git-scm.com/

[2] https://maven.apache.org/

2 Apache Flink Overview

In this section, we first describe Flink's architecture at a high level and then provide a view on how programs are built using Flink in both the batch and streaming contexts.

2.1 Architecture

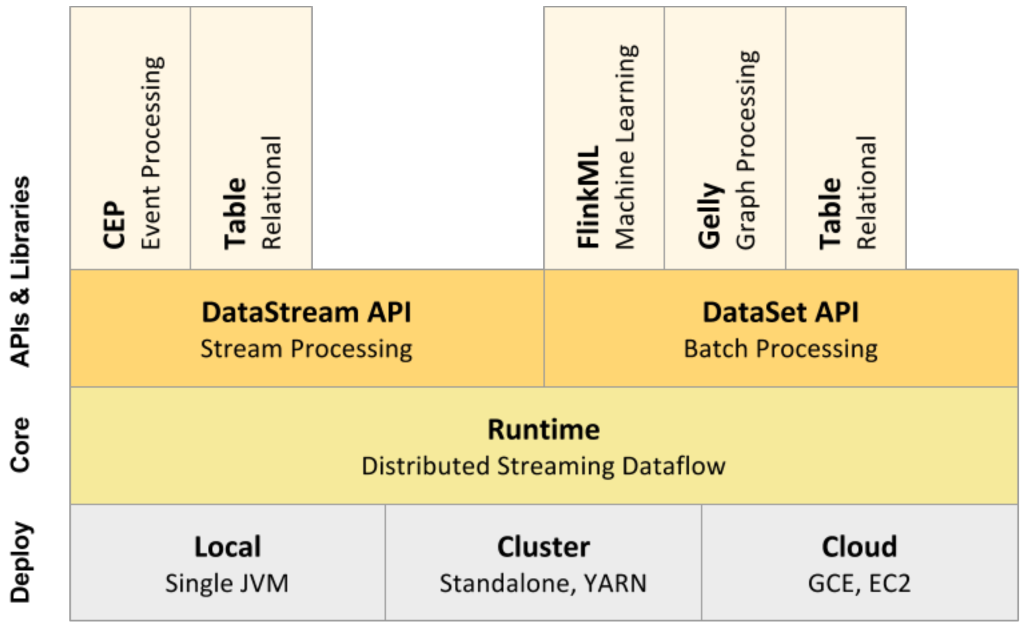

Figure 1. Architecture of Apache Flink.

Figure 1 provides an overview of the Apache Flink [5,6] architecture. The foundation of Flink is a unified runtime environment in which all programs are executed. Programs in Flink are structured as directed graphs (JobGraphs) of parallelized operations that can further contain iterations [7]. A JobGraph consists of nodes and edges. There are two classes of nodes: (stateful) operators, and (logical) intermediate results (IRs). When running a program in Flink, operators are translated into various parallel entities, which consequently process partitions of intermediate results (or input files), offering data parallelism. Unlike Hadoop, programs in Flink are not divided into individual phases that are executed sequentially (Map and Reduce). Instead all operations are executed in parallel. The results of an operator are then directly forwarded to following operator to be processed, which results in a pipelined execution. Flink programs written using one of the many APIs, as described in the next section, are translated internally into abstract data flow programs. These are then transformed into execution plans using logical and physical cost-based optimization. These can then be executed in the engine. The scheduler decides on the operator placement and tries to exploit data locality where possible.

Flink provides a distributed runtime environment for clusters and also a local runtime environment. Programs can, therefore, be run and debugged right in a local development environment easing development. The distributed engine adapts the execution plan to the cluster environment and, thus, can run different plans based on the environment and data distribution. Flink is compatible with a number of cluster management and storage solutions, like Apache Mesos, Apache Yarn, Apache Tez, Apache Hadoop, Amazon S3, Google Cloud Platform.

Stream Builder and Common API translate between the runtime environment and the interfaces (API) by transforming directed graphs of logical operations into generic data stream programs that are executed in the runtime environment. The automatic optimization of data flow programs is included in this process. The integrated optimizer for example chooses the best concrete join-algorithm for each respective used case, with the user only specifying an abstract join operation.

2.2 Libraries and Interfaces

Apache Flink users can specify their queries in various programming languages. A Scala and a Java API are available for the analysis of data streams and batch processing, respectively. Batch data can further be processed via a Python API. All APIs offer the programmer generic operators such as Join, Cross, Map, Reduce, and Filter. In this Flink differs from Hadoop MapReduce, which only allows for complex operators to be implemented as a sequence of map and reduce phases. Furthermore, users can specify arbitrary user defined functions. Listing 1 shows a word count implementation with the Scala Stream Processing API. Analogous to this example an implementation to batch process is possible with the omission of the window specification.

```scala
case class Word (word: String, frequency: Int)
val lines = env.fromSocketStream(...)
lines.flatMap{line => line.split(" ")}
   .map { (_, 1) }
   .keyBy(0)
   .timeWindow(Time.of(5, TimeUnit.SECONDS))
   .sum(1)
```

Listing 1. Word count implementation using Apache Flink's Scala stream processing API.

In the first line a tuple consisting of a string and an integer is defined. Line 2 indicates a Socket Stream, which reads a text data stream line by line. In Line 4, a FlatMap-Operator is applied, which obtains lines as input, divides these by blank spaces, and converts the resulting single words into the previously defined tuple format with the word as string and 1 as numeric value. Since this is a data stream query, a window is specified. This window is a sliding window with a duration of five seconds. Finally, the words are grouped and the numeric values are added up within the various groups. The print method outputs the result on the console. In addition to its classical interfaces the FlinkML library offers a number of algorithms and data analysis pipelines for Machine Learning (ML). Gelly enables graph analysis with Flink. The Table API allows for declarative specifications of queries similar to SQL. It is available as Java and Scala version. Listing 2 shows a word count implementation with the Java Table API for batch processing.

```java
DataSet<Word> input = env.fromElements(
   new Word("Hello", 1),
   new Word("Bye", 1),
   new Word("Hello", 1));
Table table = tableEnv.fromDataSet(input)
   .groupBy("word")
   .select("word.count as count, word");
tableEnv.toDataSet(table, Word.class).print();
```

Listing 2. A word count implementation with the Java Table API for batch processing.

Initially the input is explicitly created. Line 4 first converts the DataSet to a table to then group it according to the attribute word. Just like in SQL the select command chooses the word as well as the sums of numerators. The result table is finally converted back into a DataSet and printed.

2.3 Current Apache Flink status

Currently Flink supports two programming models: batch and streaming.

DataStream programs in Flink are regular programs that implement transformations on data streams (e.g., filtering, updating state, defining windows, aggregating). In the context of PROTEUS, the data streams are the sensor data obtained from Arcelor Mittal.

DataSet programs in Flink implement transformations on data sets (e.g., filtering, mapping, joining, grouping). In the context of PROTEUS, data sets are the historical data that Arcelor Mittal could use to detect the anomalies that happened over the past in their production plants.

Both Flink's DataStream and DataSet API can be used variety of contexts, standalone, or embedded in other programs. The execution can happen in a local JVM, or on clusters of many nodes. Table 2 shows the general differences between Flink's DataSet and DataStream API. We demonstrate the differences in detail in preceding tables. For example, operator level differences are shown at Table 3. Variations in JobGraph translation and optimization are demonstrated in Table 4. Table 5 shows the distinctions between DataSet and DataStream on Runtime and Task level semantics. Variations on scheduling are shown in Table 6 and the ones on state management are demonstrated in Table 7.

DataStream	**DataSet**
Low latency	High latency
Static Files (finite input)	Event Streams (finite input)
Pipelined Data Transfer	Blocking or Pipelined Data Transfer
Restore states are used through processing	No checkpoints
Intermediate results are not cached.	Caching of intermediate results available.

Table 2. Differences between DataStream and DataSet API in general.

Operator	**DataSet**	**DataStream**
reduce	blocking	Pipelined WindowedStream is blocking.
join	blocking	Blocking for window join
map/flatmap	pipelined	pipelined
groupBy	blocking	pipelined with operator keyBy
Checkpointing on operators	No optimizations	Incremental and asynchronous checkpointing
aggregations	structured, blocking	pipelined
union	blocking	pipelined
coGroup	blocking	pipelined

project	pipelined	pipelined
Physical partitioning	blocking	blocking

Table 3. Differences between DataSet and DataStream at the operator level.

Optimization	**DataSet**	**DataStream**
Chaining/Fusion	yes	yes
Join Strategy Selection	yes	no
Data Locality	yes	no
Window Pre-Aggregation (Panes)	n/a	yes
Iteration Head +Tail Collocation	yes	yes
Cost based optimization	yes	no

Table 4. Differences between DataSet and DataStream in the implemented graph translation and optimization.

Feature	**DataSet**	**DataStream**
Backpressure	no	yes
Barriers	no	yes
Blocking Channels (Aligning)	no	yes
Memory Allocation	Eager (Managed)	Lazy (Managed pending)
Intermediate Results	Yes	No

Table 5. Differences between DataSet and DataStream in Runtime and Task level semantics.

Feature	**DataSet**	**DataStream**

Scheduling	Lazy: Schedule tasks from sources to sinks with lazy deployment of receiving tasks.	Eager: Schedule tasks all at once instead of lazy deployment of receiving tasks.
Resource Management	Resources Released when any Task Finishes	Resources Released in Topological Order when Streams End

Table 6. Differences between DataSet and DataStream in scheduling.

Feature	**DataSet**	**DataStream**
Managed State	No	Yes (Checkpointed, KVState) Special States (ListState, MapState etc.)
Backend	File system as a materialized view	Pluggable (Memory/TM, RocksDB (out-of-core), DB (out-of-core), HDFS)
State Checkpoints	Interm. Result (I.R.) Materialisation	Periodic State Snapshot Mechanism
State Checkpointing structure	In memory/out-of-core data structures	Key-value indexes
Recovery	Recompute operator from last I.R.	Restart **job** from last global checkpoint

Table 7. Differences between DataSet and DataStrem in State management and fault tolerance.

3 Proteus hybrid computational model implementation

The hybrid data processing system implemented in this first prototype supports two kinds of inputs: historical and streaming data. In the simplest case, handling static data in streaming environment can be thought of an *operator state* (e.g., a counter or a ML model which is being trained) against which, streaming windows perform operations (e.g., updates, averaging, regression). This is the main intuition behind our computing model. As seen in Figure 1, we propose the main computational unit for hybrid system to be the Window. One can treat a dataset of historical data as a bounded Window defined over the whole history of a stream. This is a very convenient and easy to grasp concept which is used for our hybrid model.

Despite its name being hybrid, the implemented architecture consists of a single data type. That is, instead of using system architecture like in Figure 2(a), the proposed system should be a simpler and more efficient one, as is shown in Figure 2(b). Instead of having two abstractions over which programs are defined, we have to define everything over a stream processor. Again, the main intuition behind this is that datasets can be seen as windows and transformations over datasets can be implemented as transformations over windows.

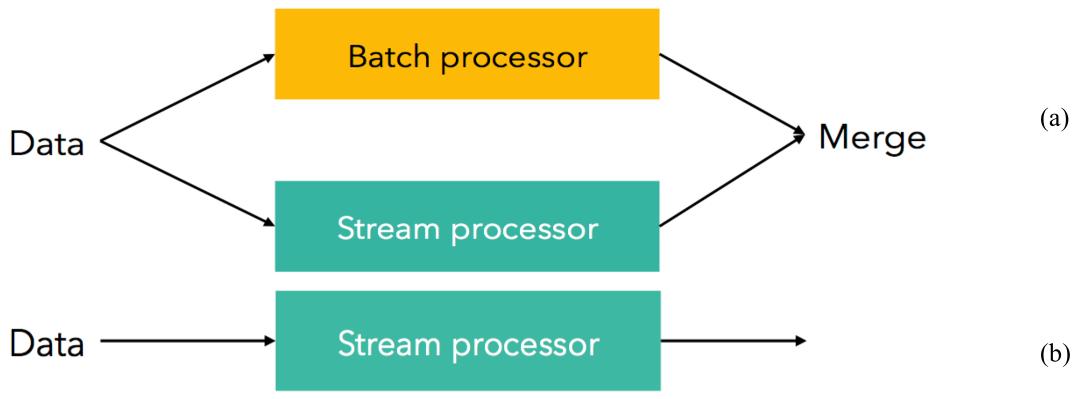

Figure 2. Different architectures to answer queries including stream and static data.

3.1 Proteus overhauled stream task processor

The key concept behind our mixed operational mode is the introduction of side inputs. This can be seen an extension of the existing broadcast sets from the DataSet API to the DataStream API. The name side input (inspired by a similar feature in Apache Beam) is preliminary but we chose to diverge from the name broadcast set because it is neither necessarily broadcast nor a set. Throughout side inputs is possible to achieve the following real-scenario tasks:

- **Join stream with static data**: a static set of data is used to filter or enrich elements in a high-throughput main stream or enrich them. Static historical data are first loaded into internal structures and then the main stream is processed. Due to the streaming setup we need to buffer arriving elements on the main input until we completely read the side input. Main input may be keyed and we also want to key the side input in order to not ship all side-input elements to all parallel instances of an operator. (This is one reason for not calling the feature broadcast stream/set).
- **Join stream with slowly evolving data**: the side input that we use for enriching or filtering the main stream is evolving over time. This feature may be implemented by buffering some initial data and then processing the main input, while continuously ingesting new side data.
- **Dynamic model update**: we would train a global model that should be used to filter or affect other elements processing. In order to properly deal with dynamic changes, we may need special triggers to continuous ingest new data into the side input data structures.

In a nutshell, a side input is a DataStream passed to a stream operator along with its main input. A different materialization policy and a custom partitioning scheme mark out side inputs.

Indeed, a side input must be materialized before than the main one in order to be used in the operator processing logic. This policy allows the stream to process the first record of the main stream input only once the side input has been fully read. Stream operators can support multiple side inputs at once and the key component that handles both main and side inputs consists of two blocking queues working according an Observer design pattern and a side inputs buffer. One blocking queue keeps track of the ready side inputs records whilst the other one stores main input records. We need blocking queues because we need to deal with inputs coming from networked nodes, thus a stream operator may ask for inputs which are not fully transferred yet. Throughout the observer pattern, Flink's I/O Manager notifies our stream operator about incoming records and consumed streams. Therefore, unless side input streams are consumed, the operator collects and materializes only their records in the side inputs buffer. At the same time, main input records are only collected. They will be materialized and processed only once side inputs are totally consumed.

A side input DataStream can be partitioned according to the following schemes:

- Broadcast: every parallel instance of the stream operator sees the whole side input as long as it can fit in memory.
- Forward: the side input is loaded from a parallel source and its shards are fed as input of the parallel stream operator instance running on the same node that actually stores the shard.
- Keyed: the side input is generated from a Keyed Stream, each keyed partition of the main input is associated to one or multiple keyed partition of the side input. A key translator is needed to map the key values of the side stream to the ones of the main input if the two streams are keyed on different types.

Through a side input, the underlying historical dataset is abstracted in a window with a dynamic granularity. As a matter of facts, it is reasonable that a big historical dataset is stored over multiple nodes using a distributed file system, therefore we may consider each shard (that will be fed to a parallel instance of the operator) as a sub-window. The number of the shards/sub-windows (and thus their sizes) depends on the parallelism degree of the operator.

Our implementation adds some latency at the beginning of the stream job because side inputs must be loaded first. Nevertheless, this will affect only the initial items ingested in the system because the side input is read only once as its updating frequency is low (if not zero). In case of slowly evolving dataset we plan to perform incremental updates to the side input to keep the latency low.

A pitfall of the current solution that we plan to address deals with making the two blocking queues fault tolerant: in this way there will be no data loss in case of nodes failures. Our solution will involve storing these records in Flink state backend.

3.2 Proteus User-level Programming API

The prototype introduces few changes to the User-level Programming Java and Scala API. First of all, we added new methods to the Streaming Execution Environment in order to let users create side inputs from already existing Data Stream.

```
public <TYPE> SideInput<TYPE> newBroadcastedSideInput(DataStream<TYPE> sideInput);
public <TYPE> SideInput<TYPE> newForwardedSideInput(DataStream<TYPE> sideInput);
public <TYPE> SideInput<TYPE> newKeyedSideInput(KeyedStream<TYPE> sideIn,
KeyTranslator kt);
```

Listing 3. Streaming Execution Environment introduced methods.

A DataStream can be linked to a side input using the following method:

```
public <TYPE, SELF extends DataStream<T>> SELF withSideInput(SideInput<TYPE> sideIn);
```

Listing 4. DataStream introduced method.

This method is based on the *Fluent interface* pattern as it improves code readability through method chaining in line with current Flink API style. This method is meant to be called on the DataStream produced by the invocation of some data transformation (i.e.: a map transform) in order to have the side input injected to the operator executing the transformation. We may introduce a persistent side input which will be injected in every transformation of the given DataStream. In this way, user may write more flexible code.

In order to lookup a side input from a *User-Defined Function*, the end-user needs to implement a Rich UDF, which has basically the permission to access some extra information besides the input itself. One of these information is the Runtime Context in which the UDF is executed. Through this Runtime Context, user normally can access to some value stored in the configuration files, broadcast datasets (only in batch API) and operator states (only in Streaming API). We extended the Runtime Context so as to let the end-user lookup a side input binded to the current transformation throughout the following method:

```
public <T> Iterable<T> getSideInput(SideInput<T> sideInput);
```

Listing 5. Runtime Context new method.

We chose the return type to be a Java iterable collection because we are most likely to deal with a side input made of several objects/tuples. On the other hand, we preferred to use a pointer to an object of type SideInput<T> because in this way we bind the generic of the return type to be the same as the side input inner type. This is needed to guarantee type safety, which would have been assured if we had used a string for the lookup

As a result of the aforementioned implementation choices, an example of what can be achieved with the current API is shown in listing 5.

```
DataStream<String> source = env.fromElements("Hello", "There", "What", "up");
DataStream<String> sideSource = env.fromElements("A", "B", "C");
SideInput<Integer> sideInput = env.newBroadcastedSideInput(sideSource);
source.map(new RichMapFunction<>() {
    public String map(String value) throws Exception {
         Iterable<Integer> side = getRuntimeContext().getSideInput(sideInput);
         LOG.info("SEEING MAIN INPUT: " + value + " with " + side);
         return value;
    }
}).withSideInput(sideInput);
```

Listing 6. An example of the new Proteus API.

4 A Machine Learning Example using Proteus API

In this section, we will show how it is possible to train an online incremental ML model using samples coming from both batch and stream sources using Proteus API illustrated in the previous section.

Before we start to introduce the example, we would like to remind the reader that our hybrid computing model is suitable for incremental ML algorithms that may benefit from accessing historical data during their training phase.

Since Apache Flink already provides a Scala ML learning library for batch datasets, the current implementation reuses as many already available software components as possible. However, as one goal of PROTEUS is to provide a set of online ML algorithms for predictions, these latter can leverage on our unified computing model, therefore the development of an online ML pipeline that naturally integrates with this model should be one of our top priority. On the other hand, we wish to clarify that we do not expect that all ML algorithms in the scope of PROTEUS have to follow this hybrid pattern, as a matter of facts, our goal is simply to offer to any data scientist the possibility to easily integrate streaming data with static ones.

In the next lines, our example shows that the fit method of the regression algorithm basically binds the side input to the main stream and apply a Rich Window Function to every window of the stream. The goal of this function is to train the desired model with the selected hyper-parameters. Moreover, we can predict samples coming from another stream using the model which is getting trained on the main input and the historical data set.

```scala
def example(args: Array[String]): Unit = {
   val env=StreamExecutionEnvironment.getExecutionEnvironment
   FlinkMLTools.registerFlinkMLTypes(env)
   val histDataSet=env.readTextFile("hdfs://mycluster:8020/user/proteus/batch.dataset")
     .map(line => parseSample(line))
   val streamingTrainingSet =env.addSource(new StreamingSource())
     .map(line => parseSample(line))
     .assignTimestampsAndWatermarks(new LinearTimestamps())
     .timeWindowAll(Time.of(500, TimeUnit.MILLISECONDS))
   val streamingTestingSet=env.addSource(new TestStreamingSource())
     .map(line => parseSample(line))
     .assignTimestampsAndWatermarks(new LinearTimestamps())
     .timeWindowAll(Time.of(500, TimeUnit.MILLISECONDS))
   val regressor=new StreamingLinearRegressionSGD()
     .withInitWeights(DenseVector.zeros(numOfFeatures), 0.0)
     .withNumIterations(100)
   val model=regressor.fit(streamingTrainingSet, env.newForwardedSideInput(histDataSet))
   regressor.predict(model, streamingTestingSet)
   env.execute()
```

Listing 7. Training and testing a LR on streaming and historical data.

5 Conclusion

The key insight of this hybrid processing model is that we provided the foundation for seamlessly integrating batch and online processing by supporting side inputs. From this important design decision, we introduced Proteus hybrid computation system, a data processing framework that combines both batch and online computations formulated in terms of Window transformations. To this end, we implemented operators supporting side inputs. A side input is a secondary input of a stream operator, that has to be materialized before processing the actual main input stream. Therefore, we changed the Flink runtime in order to handle side inputs and we exposed, to our end-user, what we call Proteus API .

The Proteus API is well integrated with the current Flink API and it lets user create new side input (for instance, from historical data set) and access them in UDF. As shown in our example, Proteus API allows fast machine learning algorithms development and quick deployment in a production environment thanks to Flink cluster managers.

References

[1] White, Tom. Hadoop: The definitive guide. " O'Reilly Media, Inc.", 2012.

[2] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

[3] https://github.com/proteus-h2020/proteus-engine

[4] Build Apache Flink from Source: https://ci.apache.org/projects/flink/flink-docs-release-1.2/setup/building.html

[5] A. Alexander, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser and V. Markl, "The stratosphere platform for big data analytics," The VLDB Journal, vol. 23, no. 6, pp. 939-964, 2014.

[6] The Apache Software Foundation, "Apache Flink," 31 12 2015. [Online]. Available: flink.apache.org. [Accessed 17 May 2016].

[7] S. Ewen, K. Tzoumas, M. Kaufmann and V. Markl, "Spinning fast iterative data flows," Proceedings of the VLDB Endowment, vol. 5, no. 11, pp. 1268-1279, 2012.