PROTEUS

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

687691

D3.1 Hybrid computation requirements definition

Lead Author: Jeyhun Karimov
With contributions from: Asterios Katsifodimos, Do Le Quoc, Alvaro Agea (LMBDP)
Reviewer: Ruben Casado (TreeLogic)

Deliverable nature:	Report (R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	31/05/2016
Actual delivery date:	30/05/2016
Version:	1.0
Total number of pages:	26
Keywords:	Hybrid computational architecture requirements

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 focus on changes that have to take place in the API and engine level of the Apache Flink platform. We first outline the requirements for analytics at the PROTEUS project and then translate those requirements to concrete changes that have to take place in Apache Flink order to be able to develop complex predictive pipelines over historical and streaming data.

Executive summary

This deliverable specifies the requirements on the Apache Flink data analytics and workflow platform which are needed to deal with analytics of streaming and historical data coming from Arcelor Mittal's (AMIII) sensors installed in their factory infrastructure. We call 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 have to 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.

The main objective of the document is to analyse the requirements and challenges for a hybrid data processing system for both historical and streaming data. We choose to build over Apache Flink, a platform which has two separate data processing models – one for streams and one for batch (historical) data. Both those processing models are executed in the same execution engine. However, the *combination* of streaming and batch analytics is not currently possible. To this end, PROTEUS will have to extend the platform and provide both the APIs which will enable hybrid computations (e.g., joining gigabytes of batch data against streams of high throughput). The requirements of the PROTEUS project are as follows:

- We need a computing model for hybrid processing of streams and batch data
- There is a need to design novel operators for hybrid processing (e.g., joins of between streams *and* historical data) which do not exist in any current processing platform.
- We need to make hybrid computing as efficient as possible in order to handle fast streams and large quantities of batch data.
- Machine learning models, and operator state needs to be supported, even when the state does not fit in main memory. Moreover, the resulting system must be fault tolerant.

Initially, we describe Flink's architecture at a high level and then we provide a view on how programs are built using Flink in both the batch and streaming contexts. Then, we define two directions to define requirements the hybrid processing system:

- We analyse the differences between DataSet and DataStream in Section 3.1. This is important when defining the requirements (in Section 3.3 and 3.4) which should support the main functionalities from both DataStream and DataSet APIs.
- We analyse the gaps in other systems with hybrid processing capability and based on that derive main requirements for our novel system in Section 3,3 and 3.4.

Afterwards, we present the extensions that should be done to the current DataStream's Window data model. These extensions require changes in:

- the programming API as seen in Listing 3
- the execution engine itself whose proposed changes are detailed Section 3.5

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**	D3.1	**Title**	Hybrid computation requirements definition
Work Package	**Number**	WP3	**Title**	Scalable Architectures for both data-at-rest and data-in-motion

Date of Delivery	**Contractual**	M06	**Actual**	M06
Status	version 1.0		final ◻	
Nature	report			
Dissemination level	public			

Authors (Partner)				
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 focus on changes that have to take place in the API and engine level of the Apache Flink platform. We first outline the requirements for analytics at the PROTEUS project and then translate those requirements to concrete changes that have to take place in Apache Flink order to be able to develop complex predictive pipelines over historical and streaming data.
Keywords	Hybrid computational architecture requirements

Version Log

Issue Date	Rev. No.	Author	Change
05/05/2016	0.1	Alvaro Agea(Novelti)	Contributions made for Section 4.
20/05/2016	0.2	Asterios Katsifodimos(DFKI)	Contributions made by Asterios Katsifodimos, Jeyhun Karimov and Le Quoc Do from DFKI.
30/05/2016	1.0	Asterios Katsifodimos (DFKI)	Minor changes based on reviews of Ruben Casado and Marcos Sacristán

Table of Contents

Executive summary 3
Document Information 5
Table of Contents 6
List of figures and/or list of tables 7
Abbreviations 8
1 Introduction 9
1.1 Document objectives 9
1.2 Document structure 10
2 Apache Flink Overview 11
2.1 Architecture 11
2.2 Libraries and Interfaces 12
3 Hybrid Data Analysis over Historical and Streaming Data 13
3.1 Current state 13
3.2 Needs and Gaps 15
3.3 A Unified Processing Model for Hybrid Computations 16
3.4 Novel operators to support computations over streams and datasets 17
3.5 Changes in the runtime engine 19
3.5.1 Current state and gaps 19
3.5.2 Requirements 21
4 State Management and Fault Tolerance 22
4.1 Management of State 22
4.2 Fault tolerance 22
5 Conclusions 24
References 25

List of figures and/or list of tables

Figure 1. Architecture of Apache Flink. 11
Figure 2. Window computational model and intuition behind its usage. 17
Figure 3. Different architectures to answer queries including stream and static data. 17
Figure 3. Design of join operator. 18
Figure 4. Example run of join operator with static and stream data. 19

Table 1. General requirements for hybrid processing system. 9
Table 2. Differences between DataStream and DataSet API in general. 13
Table 3. Differences between DataSet and DataStream at the operator level. 14
Table 4. Differences between DataSet and DataStream in the implemented graph translation and optimization. 14
Table 5. Differences between DataSet and DataStream in Runtime and Task level semantics. 14
Table 6. Differences between DataSet and DataStream in scheduling. 15
Table 7. Differences between DataSet and DataStrem in State management and fault tolerance. 15
Table 8. Existing solutions and their limitations, to form the requirements for the hybrid system. 16
Table 9. Cache operators and examples. 21

Listing 1. Word count implementation using Apache Flink's Scala stream processing API. 12
Listing 2. A word count implementation with the Java Table API for batch processing. 12
Listing 3. Similar code to the scenario shown in Figure 4. 19

Abbreviations

Acronym	Definition
API	Application Programming Interface
BDAS	Berkeley Data Analytics Stack
HDFS	(Apache) Hadoop File System
JVM	Java Virtual Machine
KVState	Key-Value State
ML	Machine Learning
SQL	Structured Query Language

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 this document tries to concentrate and define its requirements.

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 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 objectives

To build the hybrid data processing system, there are some requirements and challenges to be dealt with. The requirements and challenges defined for hybrid architecture are:

- Simple, clear, efficient, and maintainable system architecture.
- A computing model and programming API for hybrid processing.
- Operator design for hybrid processing.
- Fault tolerance.
- State management, especially when the state is too big to fit in memory.

1.2 Document structure

We will implement the hybrid computational model in Apache Flink [4]. Therefore, we first, introduce Flink in Section 2. In Section 3, we analyse the differences between DataSet and DataStream APIs of Flink in order to get requirements for our new API and execution environment. Moreover, we analyse gaps in other systems aiming hybrid computation so we can get all needs and requirements for our proposed hybrid system. Based on these analyses, we propose our computational model changes over streaming windows, and novel operator design concepts. The proposed computing model is required to overcome the distinction of static and streaming data at the system level. Furthermore, we define the required changes in Flink's execution engine to support our computational model and operator design. We talk about state management and fault tolerance in Section 4. Finally, we conclude by giving the main take-home messages of this document.

2 Apache Flink Overview

In this section we first we 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

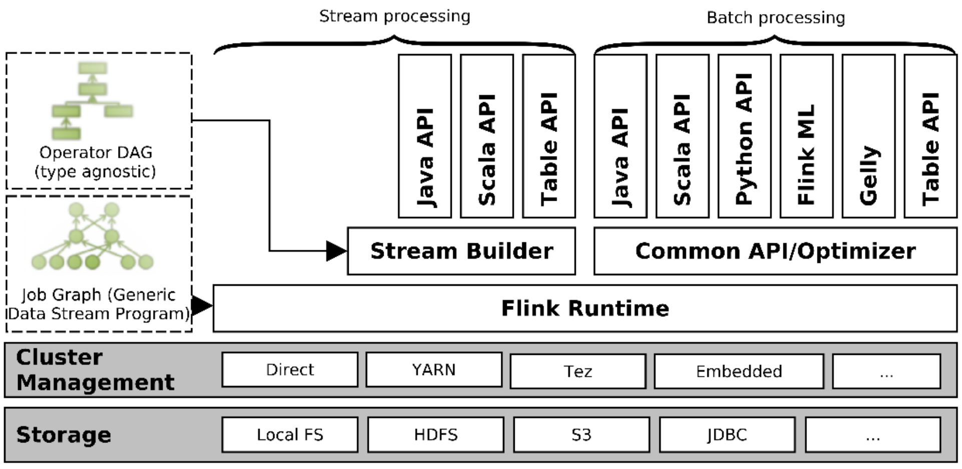


Figure 1. Architecture of Apache Flink.

Figure 1 provides an overview of the Apache Flink [3,4] 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 [5]. 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, such as Apache Tez[1], Apache Kafka[2] [10], Apache HDFS [11], and Apache Hadoop YARN [12].

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.

[1] http://tez.apache.org
[2] http://kafka.apache.org

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.

```
1 case class Word (word: String, frequency: Int)
2 val lines: DataStream[String]
3   = env.fromSocketStream(...)
4 lines.flatMap{line => line.split(" ")}
5     .map { (_, 1) }
6     .keyBy(0)
7     .timeWindow(Time.of(5, TimeUnit.SECONDS))
8     .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.

```
1 DataSet<Word> input
2    = env.fromElements(new Word("Hello",1),
3      new Word("Bye",1),new Word("Hello",1));
4 Table table = tableEnv.fromDataSet(input)
5    .groupBy("word")
6    .select("word.count as count, word");
7 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.

3 Hybrid Data Analysis over Historical and Streaming Data

In this section we present the computing model for hybrid computations over historical and streaming data.

To define the requirements for the hybrid system, we follow two directions:

- Firstly, we analyse the differences between DataSet and DataStream in Section 3.1. This is important when defining the requirements (in Section 3.3 and 3.4) for the hybrid system as it should support main functionalities from DataStream and DataSet.
- Secondly, we analyse the gaps in other systems with hybrid processing capability and based on that derive main requirements for our novel the hybrid system in Section 3,3 and 3.4.

3.1 Current state

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. As a sink, we can be write the stream to the data to files, or to standard output (for example the command line terminal).

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 machines.

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.

3.2 Needs and Gaps

In order to list the requirements for the hybrid system, we need to analyse other systems and compare their functionalities with current needs. In this way, we can get to the solution more easily.

Some systems accumulate the real time data in storage and do periodic batch jobs on top of that. Apache Flink, Apache Spark, Hadoop and etc. are good examples of batch data processing systems. However, high latency becomes major issue in this case. Moreover, there is no direct solution for hybrid (stream and batch) processing in such systems. Table 8 summarizes the needs and gaps in other systems for hybrid computation. We take those as a requirement for our system to design an efficient and novel hybrid data processing system.

Existing solution	Gaps
Summingbird[6], Lambdoop [16].	These systems based on Lambda architecture. However, they contain a problem about maintenance. They were built on several frameworks, which can be painful while maintaining them.
H-Store [8] and S-Store [7]	H-Store was implemented base on the collaboration of MIT, Brown University and the University of Washington, among others. It supports parallel query execution as well as *transactions*. S-Store, on the other hand, was developed base on the ideas and codebase of H-Store and

	was supported the feature for transactions to be executed over data streams. S-Store was designed to serve OLTP (On-Line Transactional Processing) workloads. However, this design decision makes the system scale poorly.
Apache Flink	Apache Flink is a high-performance scalable and reliable framework supporting both stream and batch processing. However, a *unified* API and execution primitives for both batch and stream processing are still missing in this framework.
Spark and BDAS	The Berkeley Data Analytics Stack (BDAS) is a framework that can serve a basis for the needs of hybrid computation. BDAS combines batch and stream processing under a common dataflow API based on resilient data bags. Spark and SparkSQL are main components of this system. One of limitations of this system, especially in our context related with ML is, it does not provide a direct access to mutable state in asynchronous manner. In addition, Spark has high latency compared with other stream processing framework, since it considers input data stream as micro-batches, and for each micro-batch a distributed data-parallel job is launched to produce the output.

Table 8. Existing solutions and their limitations, to form the requirements for the hybrid system.

3.3 A Unified Processing Model for Hybrid Computations

In his section we present the extensions that should be done to the current DataStream's Window data model. These extensions require changes in:

- the programming API (Listing 3)
- the execution engine itself (Section 3.5)

In the sequel we present the proposed operations that the DataStream API should implement in order to support hybrid computations. Instead of having separate models for batch and stream processing, we need to design a computational model such that there is only one main data structure over which, all data analysis and prediction tasks can be implemented.

The hybrid data processing system should support two types 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 can be used for our hybrid model. Moreover, by treating datasets as windows, we will have to do the least number of changes to the Apache Flink platform: we can simply take operations which are defined over datasets and enable their execution on top of windows. Moreover, any operation which includes both streams and static data can be defined over windows and data streams.

There are several points that need to be considered:

- The main computational unit (Window) should be intermediate data type between DataSet and DataStream so that it would be easy to make hybrid computations.
- The window model should inherit most of functionalities of the current streaming windows but support additional capabilities for the new hybrid operations that are described in Section 3.4.

- As can be seen from Figure 1, static data can form a relatively big window which may not be kept in memory. So, the proposed data structure is expected to handle big states that cannot be kept in memory.
- We should be able to initialize Window from streaming tuples or from static data.

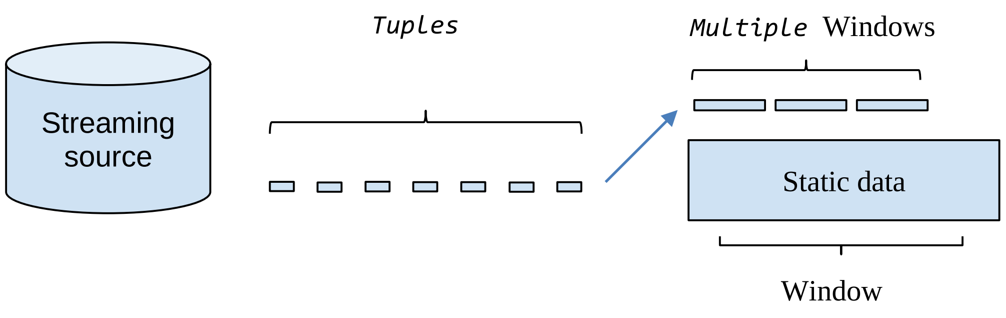


Figure 2. Window computational model and intuition behind its usage.

Prior to deciding the requirements for our unified computational model specified above, we examined other alternatives. Firstly, we analysed the conversion of DataStream window to DataSet and fire the specified operator with DataSet input. In this way, we could reuse all operators that works with DataSets and leave the DataStream API intact. However, we the DataStream and DataSet run on different execution environments and are strictly separated in Flink. Secondly, we analysed the requirements for creating new execution environment and using both DataStream and DataSet execution environments within it. This solution pretty much looks resembles the approach that we plan to take by extending the DataStream API's windows with DataSet transformations: we are moving features from the DataSet API into the DataStream API and extend the execution engine to support these changes.

3.4 Novel operators to support computations over streams and datasets

Despite its name being hybrid, the proposed architecture should consist of a single data. 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 be able 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.

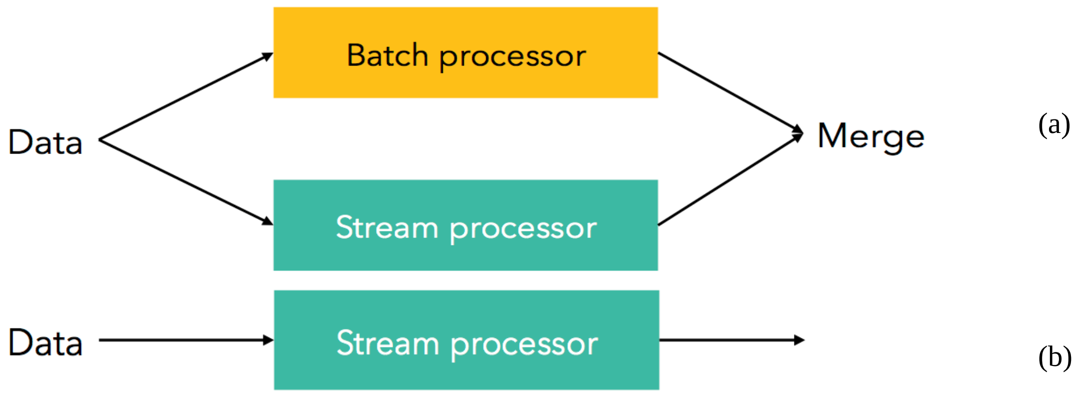


Figure 3. Different architectures to answer queries including stream and static data.

The requirements for the proposed architecture go as follows:

- The proposed architecture should be built on top of the *streaming* execution environment. Batch processing can be seen of a special case of streaming with bounded/finite streams. Thus, the proposed system is required to support all functionalities of streaming environment as well as its operators. Moreover, optimizations which are applied now on the DataSet API could be lost due to building the proposed architecture on top of stream execution environment. Although this is out of the scope of this project, we plan to investigate how one can carry the optimizations from the DataSet API to the newly proposed API which will support hyndri computations.
- The hybrid architecture should support defining operations on an easy and programmer-friendly API.
- The hybrid architecture should undergo runtime engine changes, which are shown in Section 3.5.
- Any operator of the streaming API should be usable within the hybrid execution environment. For example, we can use mappers, reducers, stream joins and *hybrid* operators in combination.

As an example, Figure 3 shows the design of a join operator such that it supports both a stream and static (historical) data.

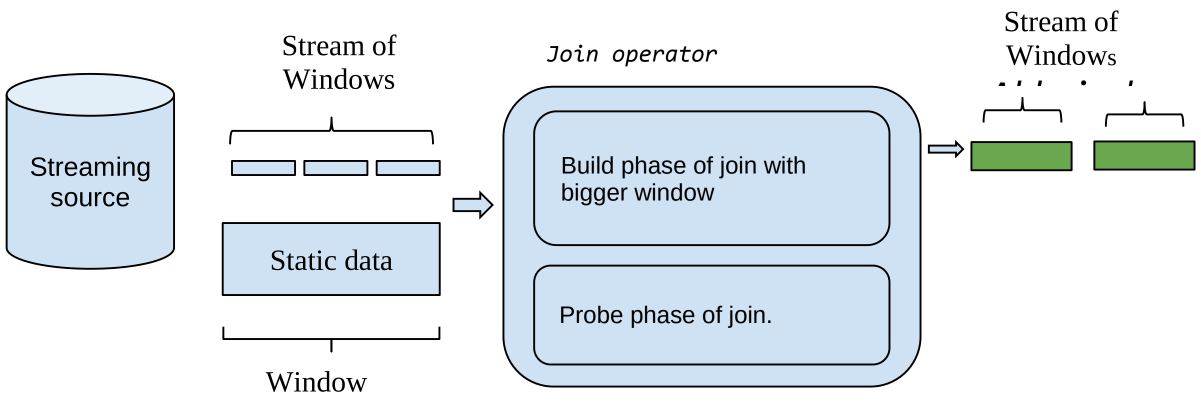
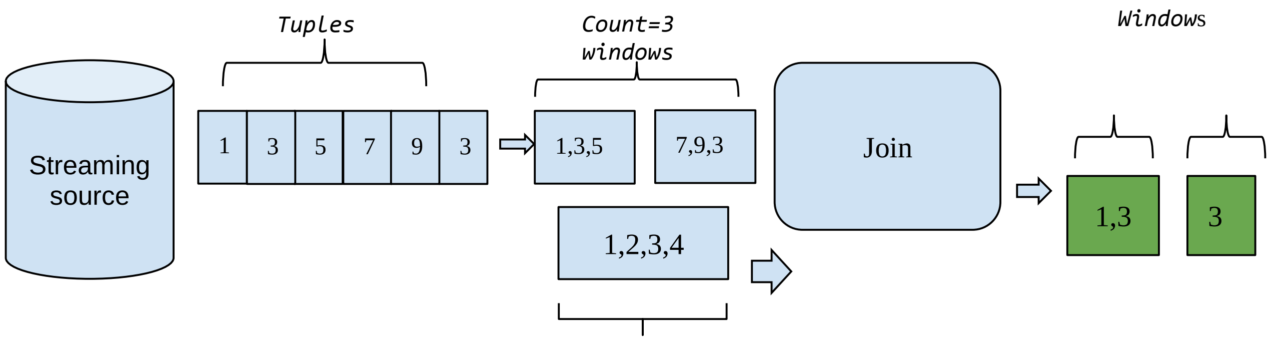


Figure 4. Design of join operator.

As we see in Figure 3, the join operator can be defined on the hybrid execution environment. Stream of windows contains windows obtained from the stream. The second Window is taken from *static data* (what is now called a *Dataset*). What we would need to do is to join together the static window along the stream of windows and get a stream of Windows as a result.

As an example, consider the scenario given in Figure 4. To join two windows in a hash based strategy, we could first build a large Window and probe the smaller Windows to get the results. The tuples are obtained from a streaming source and then divided to count windows of size 3. The aim is to join the streaming windows of size 3 against the global window of static data. As a result, we get Windows of joined data.



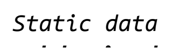


Figure 5. Example run of join operator with static and stream data.

The code of scenario could look like the one in Listing 3. There, the window is initialized from static data (directly) and from streaming data (indirectly). As a result, we get *`WindowedDataStream`* which consists of a stream of `Windows`.

```
Window<Integer> fromStaticData = env.fromElements(1,2,3,4);
DataStream<Integer> ds = env.fromElements (1,3,5,7,9,3);
WindowedDataStream<Integer>wds = ds.countWindow(3).// w1-1,3,5- w2-7,9,3
                                            join(fromStaticData).where(0).equalTo(0)
                                                                   // {wid=1,{1,3}}-{wid=2,{3}}
```

Listing 3. Similar code to the scenario shown in Figure 4.

Similar to the join, we can define transformations such as union, cross product and cogroups.

3.5 Changes in the runtime engine

In this section, we present in detail what needs to be changed in the runtime engine of Flink to support high performance hybrid computations. As described in section 3.4, our computation model for hybrid computations over historical and streaming data boils down to computations over Windows. In this model, the input arriving data items from data streams are added to a window, and the data items from historical data are also added into a window. Thereafter, the two windows are joined into an output window. This output window is processed by launching Flink jobs. These consecutive jobs could reuse intermediate results of their predecessor jobs to achieve low-latency in processing the data and avoid recomputation. Therefore, a cache mechanism is required to keep intermediate results and reuse them in subsequent jobs.

In addition, the computation window slides over the input data items, where the newly arriving input data items are added to the window and the old data items are removed from the window as they become less relevant to the analysis. There is a substantial overlap of data items between the two successive computation windows. This overlap of unchanged data-items creates an opportunity to process the data incrementally. Again, to support the incremental output update, a cache mechanism is need to keep analyzing results of windows and reuse them in the subsequent windows. However, the current runtime engine of Flink is currently missing such a mechanism. In this deliverable, we define the requirements and an initial design of the caching mechanism.

3.5.1 Current state and gaps

Currently, each time a job is submitted, Flink generates an execution graph that consists of vertices representing operators, and intermediate results representing the output produced by operators (see Figure 5). The execution graph is maintained in JobManager during the executing time.

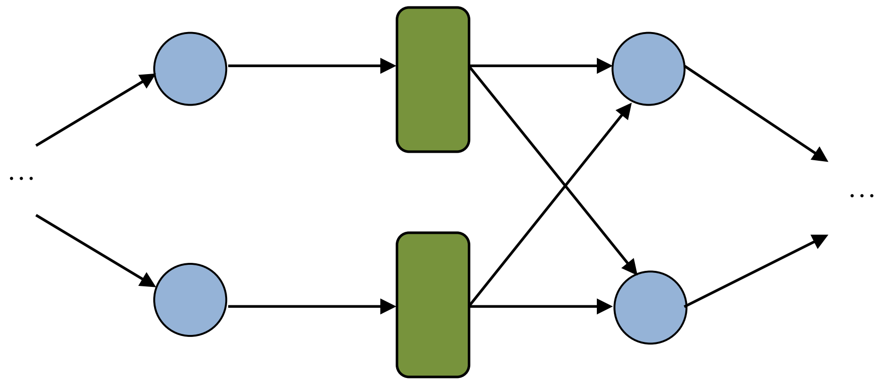


Figure 5. Flink Execution Graph.

However, the graph execution is just a logical part of the Flink job execution. The runtime part of the graph execution is responsible for the actual data execution, and is maintained by TaskManagers. The runtime part, the intermediate data is stored in blocks, called ResultPartitions i.e., a ResultPartition is a block of intermediate data produced by a task. Since data is processed in Flink in a parallel and distributed manner, a ResultPartition includes a set of ResultSubpartitions. Each ResultSubpartition is a partition of the intermediate data produced by an operator. When a ResultSubpartition is available to be consumed, a notification is generated to inform JobManager. Once the JobManager receives a notification, it schedules or updates receivers of the ResultSubpartitions. Based on the content of the notification, the Job Manager can deploy or notify consumers (see Figure 6). In Flink, there are two mechanisms for job execution: *(i)* PIPELINED and *(ii)* BLOCKING. With PIPELINED, receivers are deployed as soon as the first record is added to the ResultPartition, meanwhile with BLOCKING, receivers are deployed after the ResultPartition is done.

The life-cycle of each ResultPartition has three phases: *(i)* Produce, *(ii)* Consume, *(iii)* Release. After consuming the result partition release() function is called to clear the buffer at producing nodes. Therefore, no intermediate data is cached or maintained in the current version of Flink.

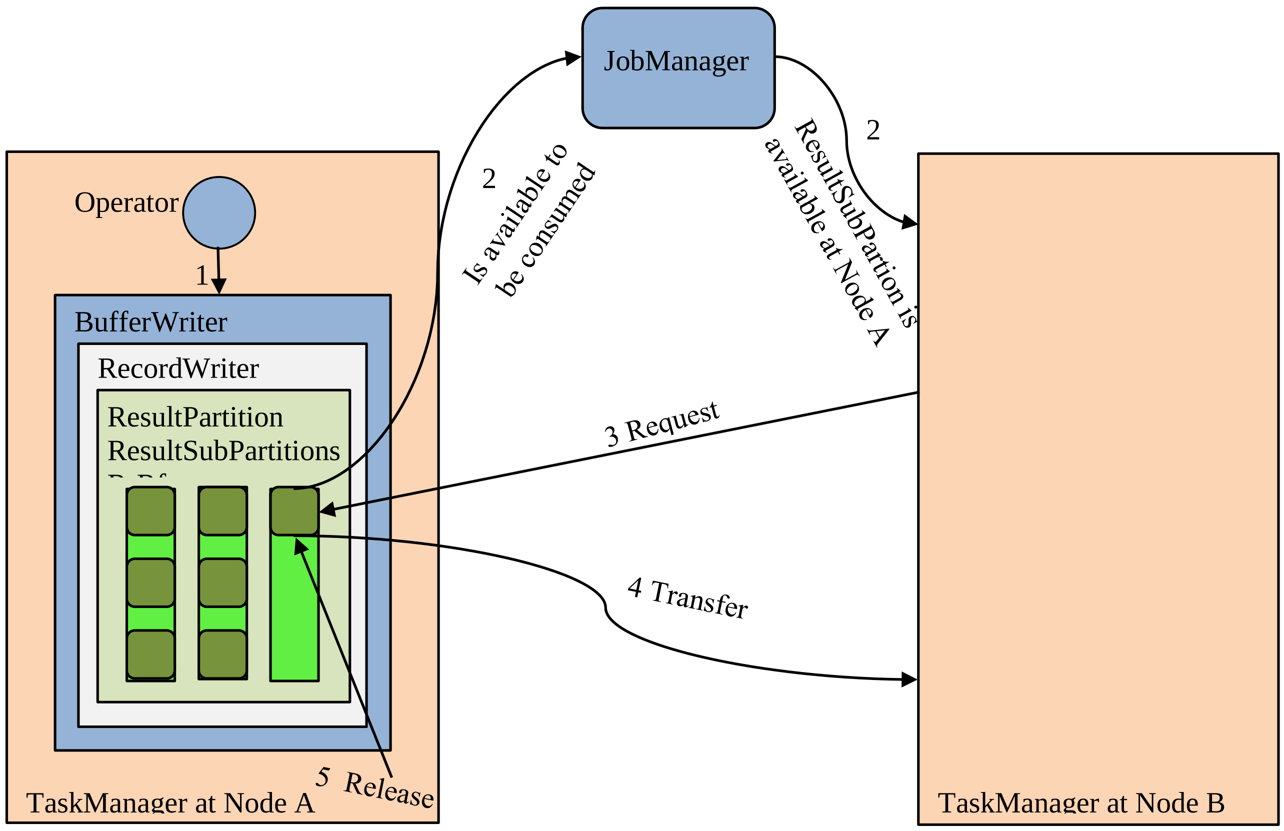


Figure 6. Producing Consuming and Releasing Intermediate Results.

3.5.2 Requirements

a. API Additions

One of the core requirements of the caching mechanism is that the cache/persist operator API must easy to use and allow the system to optimize without having the user specify execution strategies. For instance, caching could be easily done by providing a call such as *window.cache()* or *window.persist(StorageLevel).* StorageLevel is used as a flag for selecting the storage where the intermediate data is stored. The intermediate results can be persisted not only in memory, but also on hard drives and other storages: the specific storage must be defined by the input parameter StorageLevel of the persist function.

Operator	**Example**
cache() (alias for persist(**storage.MEMORY**))	window.cache()

persist(**storage.MEMORY**)
persist(**storage.DISK**)
	window.persist(**storage.MEMORY**)

unpersist()	window.unpersist()

Table 9. Cache operators and examples.

b. Intermediate data presentation

Whenever we want to reuse the intermediate data of computation windows, it calls a cache/persist operator to cache (persist) the supplied data. At this point, the cluster nodes that operate on this data must store their associated data's partitions and SubPartitions. These partitions can be later reused by window computations or jobs that later operate on the cached data. Thus, we require that the cached data must be taken into account as a data source in subsequent operations or jobs. By this way, the system can take advantage of the locality of the intermediate data, processing it near the place it is stored in order to remove the re-shuffling phase between consecutive operations.

c. Garbage Collection

In the case when a user tries to cache too much data to fit in memory, the garbage collector triggers more often and hampers Flink job's progress. The proposed cache/persist operators, if used inaptly, may stress the cache capacity and lead to an undesirable slowdown of a system. Therefore, we need an additional mechanism to release old cached data if memory capacity is not enough and recompute again in the next call, or automatically spill to disks. This mechanism ensures that the Flink job will not be broken even if user tries to cache too much data in memory. Moreover, the garbage collection can be aided by the window evictors currently supported in Flink. As a matter of fact, we plan to utilize information about expired elements of a stream in order to evict data from memory.

4 State Management and Fault Tolerance

4.1 Management of State

State management in distributed system, specially those supporting hybrid computation models, is a challenge related to maintaining the consistency of a shared state required by a set of concurrent processes. Solutions to this problem has been proposed in many fields (e.g., cache consistency in multicore architecture) and highly depend on the size of the shared state, the size of the messages sent between the different entities of the system, and the bandwidth limitations of the selected architecture.

To illustrate the problem, let us consider a classical n-body problem in two dimensions supported by a m by m matrix. In this situation, we can easily partition the matrix in such a way that each parallel stage computes the evolution of part of the space. On each evolution of the system, tasks need to update the position of the bodies outside of their assigned region in order to be able to compute the next evolution. By introducing a state management system in this scenario, task will store their updated state in this centralized system, so other task can request the information when needed.

In PROTEUS, we plan to develop a distributed ML library for hybrid computing scenarios. The algorithms to be developed in this library require the existence of a state management system. This system will provide a centralized point to manage the state of the different models and algorithms being executed.

The requirements for the hybrid processing system for the needs of PROTEUS can be summarized as follows:

- Partitioned and Global State
- Read-after-write consistency: the state management system should return the last value written on a read request.
- write linearization: Multiple write requests to the same key should be linearized to establish the write order.

4.2 Fault tolerance

Backing up and restoring the state of a computation is vital in the presence of software or infrastructure failures. On the event of such a failure, the system has to recover the state of the failed operator, and continue the computation, ideally, from where it left off. As in transaction execution, different applications require different *consistency guarantees*. For instance, if no consistency guarantee is required (e.g., a live monitoring dashboard that displays clicks in a website), a system might simply recover the operator state to any previous snapshot and continue ingesting the stream. However, a computation that requires *correctness* of results (e.g., user interactions with their shopping basket) requires stricter consistency guarantees.

The usual consistency guarantees provided by streaming platforms are:

- at most once: no consistency guarantees whatsoever, including possibility for data loss
- at least once: the fault tolerance mechanism guarantees that an event is going to update the state of an operator at least once i.e., in case of failure, some events of the stream might update the state multiple times.
- exactly once: the operator state will be updated exactly once per event, no matter how many failures may happen. Here, failure-free execution produces the same result as execution under failures

In batch processing, guaranteeing exactly once consistency is simple: the system simply pauses the computation, re-deploys (part of) the dataflow and re-consumes the necessary parts of its input. In the streaming case, however, computations are continuous (e.g., months of processing) ii) a data stream does not have, in principle, a beginning or an end, nor can it be stored indefinitely and iii) the computation cannot simply be stopped mid-flight. However, this is currently not implemented in Flink.

In the presence of state in streaming computations, the problem of providing exactly once guarantees boils down to i) determining what state the streaming computation currently is in (including in-flight records, and operator state), ii) drawing a consistent snapshot of that state, and iii) storing that snapshot in durable storage. If one can do this frequently, recovery from a failure only means i) restoring the latest snapshot from durable storage, and ii) rewinding the stream source to the point when the snapshot was taken and continuing the computation.

5 Conclusions

The key insight of hybrid processing is that we can provide the foundation for seamlessly integrating batch and online processing by converting DataSets into windows of arbitrary size. From this important design decision, we introduced the hybrid computation system, a data processing framework that combines both batch and online computations formulated in terms of Window transformations. To this end, we introduced the main concepts which are required to be implemented: a novel operations which are required in hybrid computations, along with the key changes in the Apache Flink platform in order to support them. Finally, we analysed the needs for state management and fault tolerance, concluding that very small changes should take place in that aspect.

We defined requirements for a new computation model, novel hybrid operators and required changes in runtime engine. The requirements are the following:

- Window can be the intermediate data type between DataSet and DataStream in order to define hybrid computations.
- The new Window model should inherit most of functionalities of the current streaming windows but also support additional capabilities for the new hybrid operations, which are now only defined over Datasets.
- The proposed data structure is expected to handle big states that could possibly be kept out of memory.
- The Window should support initialization from both streaming tuples and static data.

The requirements for the novel hybrid operators are the following:

- The proposed architecture should be built on the *streaming* execution environment.
- The hybrid architecture should support defining operations on an easy and programmer-friendly API.
- The hybrid architecture should undergo runtime engine changes, which are detailed in Section 3.5.
- Any operator of the streaming API should be usable within the hybrid execution environment. For example, we can use mappers, reducers, stream joins and *hybrid* operators as well as combinations of the above.

The required changes to implement caching in the execution engine go as follows:

- We need support for high level APIs for easy and programmer-friendly usage.
- An additional mechanism is needed to release old cached data if memory capacity is not enough and recompute again in the next call, or automatically spill state to disk.
- The system is required to take advantage of the locality of the intermediate data, processing it near the place it is stored in order to remove the re-shuffling phase between consecutive operations.

The next step is to get feedback from the Flink community for our design of the hybrid computing model. Based on the comments, we will define a design document and start implementation.

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] 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.

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

[5] 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.

[6] Boykin, Oscar, et al. "Summingbird: A framework for integrating batch and online mapreduce computations." Proceedings of the VLDB Endowment 7.13 (2014): 1441-1451.

[7] Meehan, John, et al. "S-Store: streaming meets transaction processing." Proceedings of the VLDB Endowment 8.13 (2015): 2134-2145.

[8] Kallman, Robert, et al. "H-store: a high-performance, distributed main memory transaction processing system." Proceedings of the VLDB Endowment 1.2 (2008): 1496-1499.

[9] R. Casado, "Lambdoop, a framework for easy development of Big Data applications," in NoSQL Matters Barcelona, 2013.

[10] Apache Kafka project.

[11] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The hadoop distributed file system," in IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010.

[12] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in Proceedings of the 4th annual Symposium on Cloud Computing, 2013.