Technical Report

The streams Framework

Version 0.9.6

Christian Bockermann and Hendrik Blom

Lehrstuhl für künstliche Intelligenz
Technische Universität Dortmund

firstname.lastname@udo.edu

technische universität dortmund

SFB 876 Verfügbarkeit von Information durch Analyse unter Ressourcenbeschränkung

Part of the work on this technical report has been supported by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", project C3.

Speaker: Prof. Dr. Katharina Morik
Address: TU Dortmund University
Joseph-von-Fraunhofer-Str. 23
D-44227 Dortmund
Web: http://sfb876.tu-dortmund.de

Contents

Part I

The *streams* Framework

Abstract

In this report, we present the *streams* library, a generic Java-based library for designing data stream processes. The *streams* library defines a simple abstraction layer for data processing and provides a small set of online algorithms for counting and classification. Moreover it integrates existing libraries such as MOA. Processes are defined in XML files following the semantics and ideas of well established tools like Ant, Maven or the Spring Framework.

The *streams* library can be easily embedded into existing software, used as a standalone tool or be used to define compute graphs that are executed on other back end systems such as the Storm stream engine.

This report reflects the status of the *streams* framework in version 0.9.6. As the framework is continuously enhanced, the report is extended along. The most recent version of this report is available online[1].

1 Introduction

In toddy's applications, data is continuously produced in various spots ranging from network traffic, log file data, monitoring of manufacturing processes or scientific experiments. The applications typically emit data in non-terminating data streams at a high rate, which imposes demanding challenges on the analysis of such streams.

We illustrate this by projects of our Collaborative Research Center SFB-876. A first example is given by the FACT telescope that is associated to project C3. This telescope observes cosmic showers by tracking light that is produced by these showers in the atmosphere with a camera. These showers last about 20 nanoseconds and are recorded with a camera of 1440 pixels at a sampling rate of 2 GHz. As about 60 of these showers are currently recorded each second, a 5-minute recording interval quickly produces several gigabytes of raw data.

Other high-volume data is produced in monitoring system behavior, as performed in project A1. Here, operating systems are monitored by recording fine grained logs of system calls to catch typical usage of the system and optimize its resource utilization (e.g. for energy saving). System calls occur at a high rate and recording produces a plethora of log entries.

The project B3 focuses on monitoring (distributed) sensors in an automated manufacturing process. These sensors emit detailed information about the furnace heat or milling pressure of steel production and are recorded at fine grained time intervals. Analysis of this data focuses on supervision and optimization of the production process.

[1] The latest version of the report is available at `http://www.jwall.org/streams/tr.pdf`.

From Batches to Streams

The traditional batch data processing aims at computations on fixed chunks of data in one or more passes. Such data is usually stored in files or database systems providing random access to each record. The results of these computations again form a fixed outcome that can further be used as input. A simple example is given by the computation of a prediction model based on some fixed set of training data. After the determination of a final model, the learning step is finished and the model is applied to deliver predictions based on the learning phase. Similar situations arise for the computation of statistics, creation of histograms, plots and the like. From a machine learning perspective, this has been the predominant approach of the last years.

Two fundamental aspects have changed in the data we are facing today, requiring a paradigm shift: The size of data sets has grown to amounts intractable by existing batch approaches, and the rate at which data changes demands for short-term reactions to data drifts and updates of the models.

The problem of big data has generally been addressed by massive parallelism. With the drop of hardware prizes and evolving use of large cloud setups, computing farms are deployed to handle data at a large scale. Though parallelism and concepts for cluster computing have been studied for long, their applicability was mostly limited to specific use cases.

One of the most influential works to use computing clusters in data analysis is probably Google's revival of the *map-and-reduce* paradigm [9]. The concept has been around in functional programming for years and has now been transported to large-scale cluster systems consisting of thousands of compute nodes. Apache's open-source *Hadoop* implementation of a map-and-reduce platform nowadays builds a foundation for various large-scale systems.

With the revival of map-and-reduce, various machine learning algorithms have been proven to be adjustable to this new (old) way of computing.

1.1 The Problem of Continuous Data

Whereas the massive parallelism addresses the batch computation of large volumes of data, it still requires substantial processing time to re-compute prediction models, statistics or indexes once data has changed. Therefore it does not fully reflect the demands for reacting to short-term drifts of data.

Within this work we will refer to this as the setting of *continuous data*, i.e. we consider an unbound source D of data that continuously emits data items d_i. In the following, we model that data stream as a sequence

$$D = \langle d_0, d_1, \ldots, d_i, \ldots \rangle$$

with $i \to \infty$. The setting to operate on streaming data is generally given by the following constraints/requirements:

C1 continuously processing *single items* or *small batches* of data,

C2 using only a *single pass* over the data,

C3 using *limited resources* (memory, time),

C4 provide *anytime services* (models, statistics).

To catch up with the requirements of large scale and continuous data, online algorithms have recently received a lot of attention. The focus of these algorithms is to provide approximate results while limiting the memory and time resources required for computation.

Analysis of Continuous Data

Traditional data analysis methods focus on processing fixed size batches of data and often require the data (or large portions of it) to be available in main memory. This renders most approaches useless for continuously analyzing data that arrives in steady streams. Even procedures like preprocessing or feature extraction can quickly become challenging for continuous data, especially when only limited resources with respect to memory or computing power are available.

At any time t we want to provide some model that reflects the analysis of the items d_i with $i \leq t$. Typical analysis tasks to compute on S are

- Given $d_i \in \mathbb{N}$ - finding the top-k most frequent values observed until t.
- For $d_i \in \mathbb{N}^p$ - find the item sets $I \subset \mathbb{N}^p$ which most frequently occurred in the d_i.
- With $d_i \subset X$, provide a classifier $c : X \to Y$, that best approximates the real distribution of labeled data $X \times Y$ (classification).
- Provide a clustering C for the data item d_i observed so far (clustering).
- Find indications on when the overall distribution of the d_i changes within the stream (concept drift detection).

Often, these tasks are further refined to models that focus on a recent sliding window of the last w data items observed, e.g. we are interested in the top-k elements of the last 5 minutes.

Algorithms for solving these tasks on static data sets exists. However, the challenging requirements in the continuous data setting are the tight limits on the resources available for computation. This can for example be real-time constraints, such as a fixed limit on the time available for processing a data item, or a bound on the memory available for computation.

Various algorithms have been proposed dedicated to computational problems on data streams. Examples include online quantile computation [12, 3], distinct counting of elements, frequent item set mining [7, 6, 8], clustering [1, 2] or training of classifiers on a stream [10].

Here, we want to provide an abstract framework for putting online learning algorithms to good use on data streams.

1.2 Designing Stream Processes

Parallel batch processing is addressing the setting of fixed data and is of limited use if data is non-stationary but continuously produced, for example in monitoring applications (server log files, sensor networks). A framework that provides online analysis is the MOA library [4], which is a Java library closely related to the WEKA data mining framework [13]. MOA provides a collection of online learning algorithms with a focus on evaluation and benchmarking.

Aiming at processing high-volume data streams two environments have been proposed by Yahoo! and Twitter. Yahoo!'s *S4* [14] as well as Twitter's *Storm* [11] framework do provide online processing and storage by building on large cluster infrastructures such as Apache's Zookeeper infrastructure.

The *Storm* engine relies on executing a computing graph, called a *topology* in Storm. The nodes (referred to as *Bolts*) and the data sources (referred to as *Spouts*) in this graph are user written programs defining the data processing and data acquisition steps. The topology is then provided by the user by implementing a Java program that creates the desired topology (a *topology builder*). To start the topology, the custom topology builder implemented by the user is given to the storm engine, which creating the bolts and distributing it along the cluster infrastructure.

While the *Storm* engine is known to be fast and scalable, it requires an in-depth knowledge of the user on how to create a topology that matches a particular task. Looking from the perspective of a *data analyst*, this does not match the higher-level rapid-prototyping needs as is adequate for domain experts e.g. in projects like telescope data analysis mentioned above.

In contrast to these frameworks, the *streams* library focuses on defining a simple abstraction layer that allows for the definition of stream processes by means of only a few basic conceptual elements. The resulting processes can be then be easily executed by the *streams* run-time or mapped to different run-time environments such as *S4* or *Storm*.

Our Contributions

In this work we introduce the *streams* library, a small software framework that provides an abstract modeling of stream processes. The objective of this framework is to establish a layer of abstraction that allows for defining stream processes at a high level, while providing the glue to connect various existing libraries such as MOA [4], WEKA [13] or the RapidMiner tool.

The set of existing online algorithms provides a valuable collection of algorithms, ideas and techniques to build upon. Based on these core elements we seek to design a process environment for implementing stream processes by combining implementations of existing online algorithms, online feature extraction methods and other preprocessing elements.

Moreover it provides a simple programming API to implement and integrate custom data processors into the designed stream processes. The level of abstraction of this programming API is intended to flawlessly integrate into existing run-time environments

like *Storm* or the RapidMiner platform [5].

Our proposed framework supports

1. Modeling of continuous stream processes, following the *single-pass* paradigm,
2. Anytime access to services that are provided by the modeled processes and the online algorithms deployed in the process setup, and
3. Processing of large data sets using limited memory resources
4. A simple environment to implement custom stream processors and integrate these into the modeling
5. A collection of online algorithms for counting and classification
6. Incorporation of various existing libraries (e.g. MOA [4]) into the modeled process.

The rest of this report is structured as follows: In Section 2 we derive a set of basic building blocks for the abstract modeling data stream processes. In Section 3 we present the XML based definition language and several addition concepts that allow for designing stream processes within the framework. Based on this we outline two example use-cases for processing and analyzing streaming data with the *streams* library. Finally we summarize the ideas behind the *streams* library and give an outlook on future work in Section 7. A comprehensive description of the implementations and guides for setting up a standalone processing environment provided by our framework is given in the appendix.

2 An Abstract Stream Processing Model

Processing streaming data can generally be viewed as establishing a *data flow graph* where each of the nodes of the graph corresponds to some function that is applied as data is passed along the edges. This general idea can be found in various existing approaches built on top of message passing systems and has been adopted by streaming systems such as SPADE, Kafka or Storm.

As the existing systems, the *streams* framework is based on an abstract definition of such data flow graphs, following the *pipes-and-filters* pattern [15]. To that it adds an additional *control flow* view, which allows for the implementation of systems fulfilling the anytime requirement as described in Section 1. Figure 1 outlines a general data flow graph built from elements like *streams* (S), *processes* (P) and *queues* (Q) and additional control flow elements represented by *services* shown in orange color.

In this section we introduce the basic concepts and ideas that we model within the *streams* framework. This mainly comprises the data flow), the control flow (anytime services) and the basic data structures and elements used for data processing. The objective of the abstraction layer is to provide a simple means for rapid prototyping of data stream processes and a clean and easy-to-use API to implement against.

The structure of the *streams* framework builds upon three aspects:

1. A *data representation* which provides a modeling of the data that is to be processed by the designed stream processes
2. Elements to model a *data flow*
3. A notion of *services* which allow for the implementation of *anytime service* capabilities.

All of these elements are provided as simple facades (interfaces) which have default implementations. The abstraction layer provided by these facades is intended to cover most of the use cases with its default assumptions, whereas any special use cases can generally be modeled using a combination of different building blocks of the API (e.g. queues, services) or custom implementations of the facades.

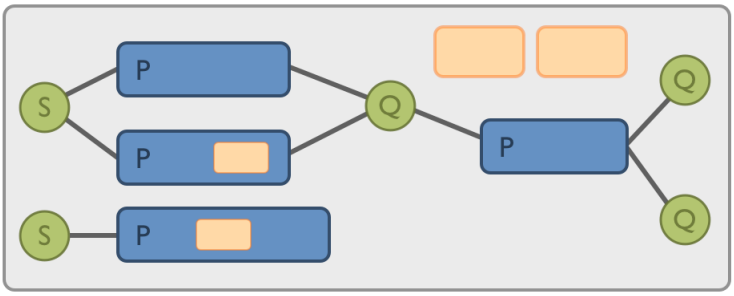


Figure 1: The general concept of a data-flow graph. The *streams* framework provides means for defining streams (S), connected processes (P), queues (Q) and adds an abstract orthogonal control flow layer (orange elements).

Executing Data Flow Graphs

The main objective of the *streams* framework is to provide an adequate abstraction layer, which enables users to design streaming processes without the need to write any code. A user can prototype *containers*, which are the top-level elements containing a data flow graph built from the base elements *stream*, *process* and *queues*. The *process* elements are the executing instances which need to be enriched with functions (*processors*) that do the actual work. The framework provides a large set of such processors that can be used to add the required functionality into the data flow graph. The containers (graphs) are defined in XML.

The XML process definitions are designed to be independent of the underlying execution platform. The *streams* framework provides its own default runtime implementation, which is able to execute the XML data flow graphs. This *streams* runtime is a Java library with a very small footprint (less than 200 kilobytes) that can be instantly executed on any Java VM.

In addition, *streams* provides a compiler to map XML process definitions to *Storm* topology, to execute processes on a Storm cluster. A third execution engine is provided for the Android platform, which allows for running *streams* definitions on mobile devices that are powered by the Android operating system.

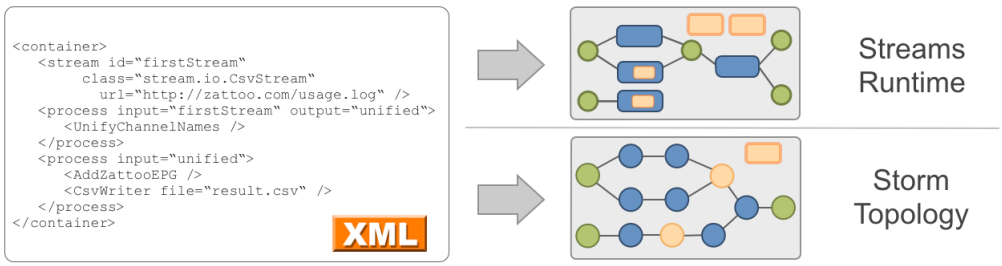


Figure 2: The XML is compiled into a data flow graph for different runtime environments. Currently the *streams* runtime is supported and a prototype for the *Storm* compiler exists.

2.1 Data Representation and Streams

A central aspect of data stream processing is the representation of data items that contain the values which are to be processed. From a message passing system point of view this is the concrete representation of messages. The abstraction of the *streams* framework considers the case of continuous streaming data being modeled as a sequence of *data items* which traverse the compute graph.

A data item is a set of (k, v) pairs, where each pair reflects an attribute with a name k and a value v. The names are required to be of type `String` whereas the values can be of any type that implements Java's `Serializable` interface. The data item is provided by the `stream.Data` interface.

Table 1 shows a sample data item as a table of (key,value) rows. This representation of data items is provided by hash tables, which are generally provided in almost every

Key	Value
x1	1.3
x2	8.4
source	"file:/tmp/test.csv"

Table 1: A data item example with 3 attributes.

modern programming language.

The use of such hash tables was chosen to provide a flexible data structure that allows for encoding a wide range of record type as well as supporting easy interoperation when combining different programming languages to implement parts of a streaming process. This enables the use of languages like Python, Ruby or JavaScript to implement custom process as we will outline in more detail in Section 5.2.

Streams of Data

A *data stream* in consequence is an entity that provides access to a (possibly unbounded) sequence of such data items. Again, the *streams* abstraction layer defines data streams as an interface, which essentially provides a method to obtain the next item of a stream.

Figure 3: A *data stream* as a sequence of *data item* objects.

The core *streams* library contains several implementations for data streams that reveal data items from numerous formats such as CSV data, SQL databases, JSON or XML formatted data. A list of the available data stream implementations is available in the appendix B.1.

In addition, application specific implementations for data streams can easily be provided by custom Java classes, as is the case in the FACT telescope data use-case outlined in section 6.1.

2.2 Processes and Processors

The *data streams* defined above encapsulate the format and reading of data items from some source. The *streams* framework defines a *process* as the consumer of such a source of items. A process is connected to a stream and will apply a series of *processors* to each item that it reads from its attached data stream.

Each *processor* is a function that is applied to a data item and will return a (modified or new) data item as a result. The resulting data item then serves as input to the next processor of the process. This reflects the pipes-and-filters concept mentioned in the beginning of this section.

The *processors* are the low-level functional units that actually do the data processing and transform the data items. There exists a variety of different processors for manipulating data, extracting or parsing values or computing new attributes that are added to the data items. From the perspective of a process designer, the *stream* and *process* elements form the basic data flow elements whereas the processors are those that do the work.

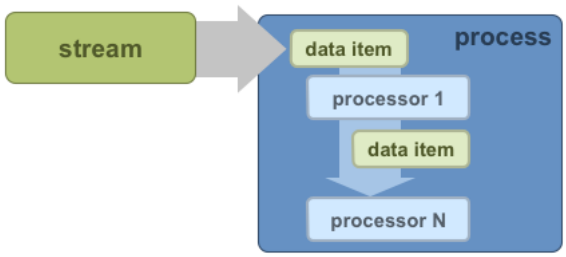


Figure 4: A process reading from a stream and applying processors.

The simple setup in Figure 4 shows the general role of a process and its processors. In the default implementations of the *streams* library, this forms a *pull oriented* data flow pattern as the process reads from the stream one item at a time and will only read the next item if all the inner processors have completed.

Where this pull strategy forms a computing strategy of *lazy evaluation* as the data items are only read as they are processed, the *streams* library is not limited to a *pull oriented* data flow.

Using multiple Processes

In the *streams* framework, processes are by default the only executing elements. A process reads from its attached stream and applies all inner processors to each data item. The process will be running until no more data items can be read from the stream (i.e. the stream returns `null`). Multiple streams and processes can be defined and executing in parallel, making use of multi-core CPUs as each process is run in a separate thread[2].

For communication between processes, the *streams* environment provides the notion of

[2]This is the default behavior in the reference *streams* runtime implementation. If *streams* processes are executed in other environments, thus behavior might be subject to change.

queues. Queues can temporarily store a limited number of data items and can be fed by processors. They do provide stream functionality as well, which allows queues to be read from by other processes.

Figure 5 shows two processes being connected by a queue. The enlarged processor in the first process is a simple *Enqueue* processor that pushes a copy of the current data item into the queue of the second process. The second process constantly reads from this queue, blocking while the queue is empty.

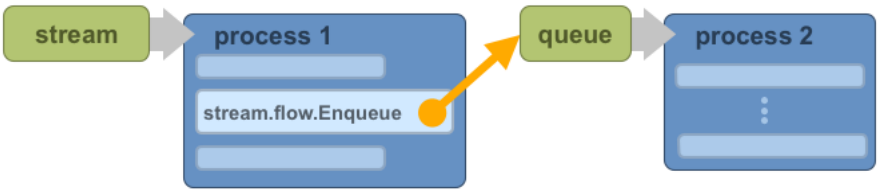


Figure 5: Two Processes P_1 and P_2 communicating via queues.

These five basic elements (*stream*, *data item*, *processor*, *process* and *queue*) already allow for modeling a wide range of data stream processes with a sequential and multi-threaded data flow. Apart from the continuous nature of the data stream source, this model of execution matches the same pipelining idea known from tools like RapidMiner, where each processor (operator) performs some work on a complete set of data (example set).

2.3 Data Flow and Control Flow

A fundamental requirement of data stream processing is given by the *anytime paradigm*, which allows for querying processors for their state, prediction model or aggregated statistics at any time. We will refer to this anytime access as the *control flow*. Within the *streams* framework, these anytime available functions are modeled as *services*. A service is a set of functions that is usually provided by processors and which can be invoked at any time. Other processors may consume/call services.

This defines a control flow that is orthogonal to the data flow. Whereas the flow of data is sequential and determined by the data source, the control flow represents the anytime property as the functions of services may be called asynchronous to the data flow. Figure 6 shows the flow of data and service access.

Examples for services my be classifiers, which provide functions for predictions based on their current state (model); static lookup services, which provide additional data to be merged into the stream or services that can be queried for current statistical information (mean, average, counts).

Service References and Naming Scheme

In order to define the data flow as well as the control flow, a naming scheme is required. Each service needs to have an unique identifier assigned to it. This identifier is available

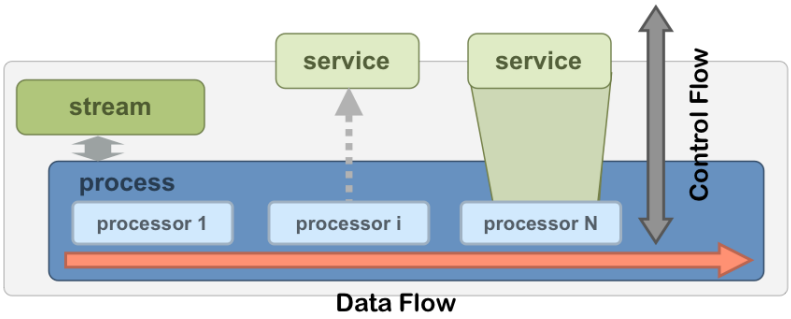


Figure 6: Orthogonal *data* and *control flow*. Processors may use services as well as export functionality by providing services.

within the scope of the experiment and will be used by service consumers (e.g. other processors) to reference that service.

At a higher level, when multiple experiments or stream environments are running in parallel, each experiment is associated with an identifier by itself. This imposes a hierarchical namespace of experiments and services that are defined within these experiments. The *streams* library constitutes a general naming scheme to allow for referencing services within a single experiment as well as referring to services within other (running) experiments.

A simple local reference to a service or other element (e.g. a queue) is provided by using the identifier (string) that has been specified along with the service definition. Following a URL like naming format, services within other experiments can be referenced by using the experiment identifier and the service/element identifier that is to be referred to within that experiment, e.g.

`//experiment-3/classifier-2`.

Such names will be used by the *streams* library to automatically resolve references to services and elements like queues.

3 Designing Stream Processes

The former section introduced the main conceptual elements of the *streams* library for creating data flow graphs. Such graphs are contained within a *container*. A container can be deployed by compiling the container definition into a data flow graph (or *compute graph*) for the runtime environment which is to execute the container.

Each of the basic elements for the process design (i.e. container definition) directly correspond to an XML element that is used to define a node in the data flow graph. The following Table 2 lists the base elements provided.

Graph Element	XML Element
Stream	`stream`
Process	`process`
Queue	`queue`
Service	`service`

Table 2: The basic XML element used to define a compute graph within the *streams* framework.

The definition of stream processes is based on simple XML files that define processes, streams and queues by XML elements that directly correspond to the elements presented in Section 2. Figure 1 shows the scheme of mapping the XML process definitions into data flow graphs of the *streams* runtime.

In addition there exists mappings for other runtime environments.

3.1 Layout of a Process Environment

As mentioned above, the top-level element of a *streams* process definition is a *container*. A single container may contain multiple processes, streams and services, which are all executed in parallel. An example for a container definition is provided in Figure 7.

```
<container id="example">
    <stream id="D" url="file:/test-data.csv" />

    <process input="D">
        <!--
            The following 'PrintData' is a simple processor that outputs each
            item to the standard output (console)
          -->
        <stream.data.PrintData />
    </process>
</container>
```

Figure 7: A simple container, defining a stream that is created from a CSV file.

The core XML elements used in the simple example of Figure 7 are `stream` and `process`, which correspond to the conceptual elements that have previously been introduced in Section 2 and which are mapped to XML elements according to Table 2. This example defines a container with namespace `example` which corresponds to the simple compute graph shown in Figure 8.

Figure 8: The simple compute graph that is defined by the XML given in Figure 7.

The graph contains a single source of data (*stream*) and only one *process* element, which consumes the data provided by the stream and applies the nested processor `PrintData` to each data item obtained from the stream.

3.1.1 Defining a Stream Source

As you can see in the example above, the `stream` element is used to define a stream object that can further be processed by some processes. The `stream` element requires an `id` to be specified for referencing that stream as input for a process.

In addition, the `url` attribute is used to specify the location from which the data items should be read by the stream. There exists Java implementations for a variety of data formats that can be read. Most implementations can also handle non-file protocols like `http`. The class to use is picked by the extension of the URL (`.csv`) or by directly specifying the class name to use:

```
<stream  id="D" class="stream.io.CsvStream"
        url="http://download.jwall.org/stuff/test-data.csv" />
```

Figure 9: Defining a stream that reads from a HTTP resource.

Additional stream implementations for Arff files, JSON-formatted files or for reading from SQL databases are also part of the *streams* library. These implementation also differ in the number of parameters required (e.g. the database driver for SQL streams). A list of available stream implementations can be found in Appendix B.1. The default stream implementations also allow for the use of a `limit` parameter for stopping the stream after a given amount of data items.

3.1.2 A Stream Process

The `process` element of an XML definition is associated with a data stream by its `input` attribute. This references the stream defined with the corresponding `id` value. Processes may contain one or more *processors*, which are simple functions applied to each data item as conceptually shown in 2.2.

A process will be started as a separate thread of work and will read data items from the associated stream one-by-one until no more data items can be read (i.e. `null` is returned by the stream). Processes in the *streams* framework are following greedy strategy, reading and processing items as fast as possible.

The processes will apply each of the nested processors to the data items that have been read from the input. The processors will return a processed data item as result, which is in turn the input for the next processor embedded into the process. Thus, each process applies a pipeline of processors to each data item. If any processor of the pipeline returns `null`, i.e. no resulting data item, then the pipeline is stopped and the process skips to reading the next data item from the stream.

The inner processors of a process are generally provided by Java implementations and are represented by XML elements that reflect their Java class name.

In the example in Figure 10, a processor implemented by the class `my.package.MyProcessor` is added to the process. The process in this examples is attached to the stream or queue defined with ID `id-of-input`. Any output of that process that is not `null`, will be inserted into the queue with ID `queue-id`. Connecting the output of a process to a queue (which can then be the input to another processor) is optional.

```
<process input="id-of-input" output="queue-id">
    <!--
         One or more processor elements, referenced
         by their class name, provided with attributes
       -->
    <my.package.MyProcessor param="value" />
</process>
```

Figure 10: A process references an input (i.e. a *stream* or a *queue*) and contains a list of processor elements. Optionally it feeds results to an associated output (a *queue*).

3.1.3 Processing Data Items

As mentioned in the previous Section, the elements of a stream are represented by simple tuples, which are backed by a plain hashmap of keys to values. These items are the smallest units of data within the *streams* library.

The smallest *functional* units of the *streams* library are provided by simple *processor*s. A *processor* is essentially a function that is applied to a data item and which returns a data item (or `null`) as result as shown in the *identity* function example below.

```
public Data process( Data item ){
    return item;
}
```

Figure 11: The `process(Data)` method - unit of work within *streams*.

Processors are usually implemented as Java classes, but can be provided in other (scripting) languages as well. The Java classes are expected to follow the JavaBeans specification by providing `get`- and `set`-methods for properties. These properties in turn will be mapped to XML attributes of the corresponding XML element. This allows processors to be easily provided with parameters wihtin the XML container definitions.

3.2 Parameterising Containers

The general structure of container definitions described in Section 3.1 allows for the definition of compute graphs and adding processors and parameters. For a convenient parameterization, the *streams* framework supports the global definition of properties and includes a intuitive variable expansion, following the syntax of well known tools like Ant and Maven.

Variables are specified using the `$` symbol and curly bracket wrapped around the property name, e.g. `${myVar}`. This directly allows to access the Java VM system properties within the container definition. Undefined variables simple resolve to the empty string.

```
<container>

    <!-- define property 'baseUrl' using the system property 'user.home'   -->
    <property name="baseUrl" value="file:${user.home}/data/FACT" />

    <stream id="factData" class="fact.io.FactEventStream"
           url="${baseUrl}/example-data.gz" />

    <process input="factData">
       <!-- process the data  -->
    </process>
</container>
```

Figure 12: A container definition using simple variables.

As the variable expansion includes the Java system properties, containers can easily be provided with variables by setting properties when starting the Java system. The following commands starts the *streams* runtime with a container definition and adds addition variables:

```
java -DbaseUrl="/tmp" -cp stream-runner.jar container.xml
```

Variables can be used anywhere in the XML attributes, the variables of a container are expanded at startup time. Therefore any changes of the variables after the container has been started will not affect the configuration.

4 Machine Learning with Continuous Data

As the large volumes of data are merely managable with automatic processing of that data, they are far away from being inspected manually. On the other hand gaining insight from that data is the key problem in various application domains.

Machine learning and data mining has put forth a plethora of techniques and algorithms for pattern recognition, learning of prediction model or clustering that all aim at exactly that key problem: knowledge discovery from data.

For the setting of continuous data, various algorithms have been proposed which solve basic tasks inherent to the knowledge discovery process as well as complex methods that allow for training classifiers or finding clusters on steady streams of data. In this section we will give an overview of how machine learning algorithms are embedded into the *streams* framework using a simple Naive Bayes classifier as example.

We first give an overview of the data representation that is used for learning from the *data items* that represent the basic data format of *streams*. Following that, we outline how learning algorithms are embedded into the framework of continuous processes. Here we provide an example for online classification and the computation of statistics.

Based on the embedding of online learning schemes for classification, we show the integration of the MOA library into the *streams* framework, which allows for directly using the set of existing classifiers for learning (Section 4.2).

The evaluation of online learning often requires large amounts of data. In Section 4.3 we show how to generate synthetic data streams for testing online learning algorithms.

4.1 Online Learning from Data Streams

The general definitions of learning tasks in online learning do not differ from the traditional objectives. Supervised learning such as classification or regression tasks rely on a source of training data to build models that can then be applied to new data for prediction.

Learning from unbounded and continuous data imposes demanding challenges to the designer of machine learning algorithms. Even simple basic building blocks like the computation of a median or minima/maxima values that might be required in a learning algorithms tend to become difficult.

4.1.1 Learning from Data Items

Online learning algorithms usually require a data representation similar to batch learning methods. Typically instances or examples used for learning are tuples of some real-valued or finite space.

As an example, the task of (binary) classification can be stated as estimating a function $\hat{f}$ that best approximates a true (unknown) distribution of instances (x, y) where $x \in M^r$ and $y \in \{-1, 1\}$. Usually features are encoded such that $M = \mathbb{R}$ in many application

domains.

In the *streams* framework we encode each of these tuples as data items by defining a key k_i for each dimension of M^p and a special key for the label y. By convention, special keys are prefixed with an `@` character. These special keys are expected to be ignored as attributes by any learning algorithm. Figure 13 shows an instance of learning that is represented by a data item.

$$(x, y) \overset{\text{e.g.}}{=} (0.3, 0.57, \ldots, 0.413, -1) \quad \rightarrow$$

Key	x1	···	xp	@label
Value	0.3	···	0.413	-1

Figure 13: Data item representation of an instance for learning, key/value table transposed for brevity.

As the attributes may hold any `Serializable` values, a proper pre-processing might be required for applying learning algorithms, e.g. if these algorithms cannot handle arbitrary data types. Such preprocessing is for example a String-to-Number conversion (provided by the `ParseDouble` processor). The *streams* core classes provide a wide number of preprocessing processors.

Filtering Attributes

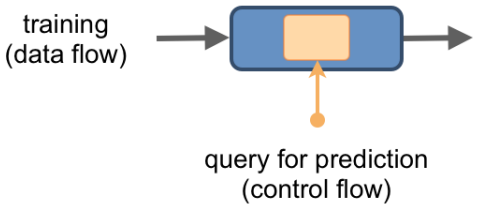
Sometimes it is desirable to train a classifier only on a subset of the features/attributes that are contained in the data. The `WithKeys` processor, allows for the execution of nested processors on filtered data items. As an example, the XML snippet in Figure 14 shows the data preprocessing to apply online learning to the famous Iris data set with only two of the attributes being selected.

```
...
<process input="iris">

    <!-- Rename the "class" attribute to "@label" as by convention
         a learner expects the label in attribute "@label"        -->
    <Rename from="class" to="@label" />

    <!-- select two attributes and the label from the
         data items and apply the inner processors />
    <WithKeys keys="att1,att2,@label">

        <!-- parse the attributes att1 and att2 to Double values -->
        <ParseDouble keys="att1,att2" />

        <!-- feed the data item to a naive bayes classifier for training -->
        <stream.classifier.NaiveBayes id="myNaiveBayes" />
    </WithKeys>

</process>
```

Figure 14: Example XML for training a classifier on a subset of attributes.

4.1.2 Embedding Classifiers in *streams*

From a high level viewpoint, an online classifier is able to continuously learn on training data and give out predictions for instances based on its current model/state. The following figureshows the division of this behavior into two parts:



This abstract functionality provided by classifiers from the perspective of the *streams* framework encapsulates two functions:

1. **Training:** Incorporate new data items into a prediction model.
2. **Application:** Provide a prediction for a data item based on the current model.

These two tasks are mapped onto two different aspects of the compute graph that builds the basis for the streaming processes. The *training* is considered to be part of the general data flow, i.e. data items are processed by classifiers and will be used to enhance the prediction model provided by the classifier.

The model *application*, i.e. the prediction based on the current model of the classifier, is regarded as an *anytime service* that is provided by the classifier. This service provides a `predict(Data)` function that is expected to return the prediction of the classifier.

Providing Predictions at any time

With the requirements for continuous data that we introduced in the very beginning, the main concern is, that classifiers need to be able to provide predictions at any time. With the *control flow* layer provided by the notion of services, the *streams* framework integrates a tool for querying services from "outside", i.e. not using the data flow. Such services are registered within the container and can be queried from other containers or being exported (e.g. via Java RMI).

```
public interface Classifier extends Service {
    /**
     * This method returns a simple prediction based on the given
     * data item. The prediction is a general serializable value.
     */
    public Serializable predict( Data item );
}
```

Figure 15: The `Classifier` service interface that needs to be implemented by classifiers in the *streams* framework. The return type of the `predict` method might be a number, e.g. for regression or a String, Integer or similar for a classification task.

4.1.3 Using the Classifier service

The example in Figure 14 shows the use of a classifier that is added to the data flow for constant training. The referenced `NaiveBayes` class implements the `Processor` interface and additional supports the `Classifier` interfaces shown in 15. The `id` of the classifier (processor) specifies the name under which that classifier is registered within the naming service of the container.

After initialization time, the classifier is registered and can be queried using Java RMI or by other processors from within the container (or connected containers). This is illustrated in the following abstraction (Figure 16): The process within that graph contains a processor that provides a service (marked in orange) and another process that references that services, i.e. plays the role of a *service consumer*.

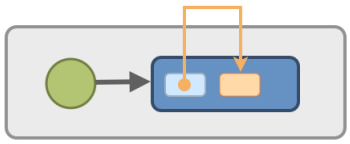


Figure 16: A processor using a processor that provides a services (e.g. Classification).

Adding Predictions

One of the processors that can be used to add predictions based on classifier services is the `AddPrediction` class. This processor references a classifier service by its `id` and calls the `predict(Data)` method for every item it processes.

The prediction of the classifier is then added to the data item with an attribute key `@prediction:` followed with the `id` of the service. An example is given in the XML snippet shown in Figure 17. The `AddPrediction` element will use the classifier for prediction and then add the prediction as a special attribute with key `@prediction:NB` to the data item. Since special attributes are to be discarded by the learning algorithms by convention, it will not influence the training.

```
...
<process input="golf">
    <!-- pre-processing left out for brevity    -->

    <!-- apply a prediction to each data item based
         on the specified classifier               -->
    <stream.learner.AddPrediction classifier="NB" />

    <!-- consume the next data item and use it for
         learning.                                 -->
    <stream.classifier.NaiveBayes id="NB" />
</process>
```

Figure 17: First predict, then learn on a data item.

4.2 Integrating MOA

MOA is a software package for online learning tasks. It provides a large set of clustering and classifier implementations suited for online learning. Its main intend is to serve as an environment for evaluating online algorithms.

The *streams* framework provides the `stream-analysis` artifact, which includes MOA and allows for integrating MOA classifiers directly into standard stream processes. This is achieved by wrapping the data item processed in the *streams* framework into instances required for MOA. Additionally, a generic class wraps all the MOA classifier implementations into a processor that implements the `Classifier` interface. MOA classifiers will be automatically discovered on the classpath using Java's reflection API and will be added to the processors available.

The following example XML snippet shows the use of the Naive Bayes implementation of MOA within a *streams* container. The example defines a standard test-then-train process.

```
<container>
     <stream id="stream" class="stream.io.CsvStream"
             url="classpath:/multi-golf.csv.gz" limit="100"/>

     <process input="stream">
          <RenameKey from="play" to="@label" />

          <!-- add  @prediction:NB based on the classifier "NB"  -->
          <stream.learner.AddPrediction classifier="NB" />

          <!-- compute the loss for all attributes starting with @prediction:
               and add a corresponding @error: attribute with the loss   -->
          <stream.learner.evaluation.PredictionError />

          <!-- incorporate the data item in to the model (learning)  -->
          <moa.classifiers.bayes.NaiveBayes id="NB"/>

          <!-- incrementally group the @error:NB   -->
          <stream.statistics.Sum keys="@error:NB" />
     </process>
</container>
```

Figure 18: Test-then-train evaluation of the MOA Naive Bayes classifier using the `AddPrediction` processor and the `Sum` processor to sum up the prediction error.

4.3 Synthetic Data Stream Generator

Testing online algorithms often requires a large amount of data that matches a known distribution or can be designed such that specific test-cases can be created for algorithms.

The *streams* core package already defines a set of streams for random data generation. In Combination with the concept of MultiStreams this can easily be used to create tailored data streams.

4.3.1 Example: A Gaussian Stream

The `stream.generator.GaussianStream` class implements a data stream that generates an unlimited sequence of normal distributed data. The default setup focuses on a single attribute with a mean of 0.0 and a standard deviation of 1.0:

```
<stream id="gauss" class="stream.generator.GaussianStream" />
```

Using the `attributes` parameter allows to specify the mean and standard deviation of one or more attributes:

```
<stream id="gauss-2" class="stream.generator.GaussianStream"
        attributes="0.0,1.0,2.0,0.25,8.5,2.75" />
```

The `gauss-2` stream above produces a sequence of data items each of which holds attributes `x1`, `x2` and `x3` based on the following distributions:

Attribute	Mean	Standard Deviation
x1	0.0	1.0
x2	2.0	0.25
x3	8.5	2.75

The attributes are named `x1`, `x2` and `x3` but can be named according to a preset using the `keys` parameter of the `GaussianStream` class:

```
<stream id="gauss-2" class="stream.generator.GaussianStream"
        attributes="0.0,1.0,2.0,0.25,8.5,2.75"
        keys="A,B,C" />
```

4.3.2 Example: A cluster data-stream

The stream `gauss-2` from above will create a sequence of data items which are centered around (0.0,2.0,8.5) in a 3-dimensional vector space.

By combining the concept of *Multistreams* with the gaussian streams, we can easily define a stream that has multiple clusters with pre-defined centers. The `RandomMultiStream`

class is of big use, here: It allows for randomly picking a substream upon reading each item. The picks are uniformly distributed over all substreams.

The following definition specifies a stream with data items of 4 clusters with cluster centers (0.0,0.0), (1.0,1.0), (2.0,2.0) and (3.0,3.0):

```
<stream id="clusters" class="stream.io.multi.RandomMultiStream">

    <stream id="cluster-1" class="stream.generator.GaussianStream"
            attributes="1.0,0.0,1.0,0.0" />

    <stream id="cluster-2" class="stream.generator.GaussianStream"
            attributes="2.0,0.0,2.0,0.0" />

    <stream id="cluster-3" class="stream.generator.GaussianStream"
            attributes="3.0,0.0,3.0,0.0" />

    <stream id="cluster-4" class="stream.generator.GaussianStream"
            attributes="4.0,0.0,4.0,0.0" />
</stream>
```

4.3.3 Imbalanced Distributions

In some cases a unified distribution among the sub-streams is not what is required. The `weights` parameters lets you define a weight for each substream, resulting in a finer control of the stream. As an example, the `weights` parameter can be used to create a stream with a slight fraction of outlier data items:

```
<stream id="myStream" class="stream.io.multi.RandomMultiStream"
        weights="0.99,0.01">

    <stream id="normal" class="stream.generator.GaussianStream"
            attributes="1.0,0.0,1.0,0.0" />

    <stream id="outlier" class="stream.generator.GaussianStream"
            attributes="2.0,0.0,2.0,0.0" />
</stream>
```

In this example, approximately 1% of the data items is drawn from the outlier stream, whereas the majority is picked from the "normal" stream.

5 Extending the *streams* Framework

In the previous sections we outlined how to create data flow graphs for data stream processing by means of the XML elements that are provided by the *streams* framework. We also introduced the *processor*s as the atomic functional units that provide the work required to do the data processing.

The core *streams* framework provides a rich set of basic processors that can be used to aggregate statistics, manipulate data and even incorporates existing libraries such as the *MOA* online learning library (see Section 4.2 for details on using MOA with *streams*).

In many application use cases, we still face the problem of requiring application specific pre-processing or functionality that cannot directly be achieved with the existing *streams* core processors. Such functionality can easily be added by custom implementations of processors. The *streams* framework provides a very simply Java API which encapsulates the abstract concepts outlined in Section 2.

In this section, we will highlight the general ideas of the *streams* programming API and provide a walk-through on how to implement custom processors using the Java language.

As pointed out earlier, the *streams* framework also provides support for scripting languages, such as JavaScript, Ruby or Python. In Section 5.2 we will give an overview on how to implement custom processors using such scripting languages.

5.1 Implementing Custom Processors

Processors in the *streams* framework can be plugged into the processing chain to perform a series of operations on the data. A processor is a simple element of work that is executed for each data item. Essentially it is a simple function:

```
public Data process( Data item ){
     // your code here
     return item;
}
```

The notion of a processor is captured by the Java interface `stream.Processor` that simply defines the `process(Data)` function mentioned above:

```
public interface Processor {
    public Data process( Data item );
}
```

Figure 19: The interface that all processors need to implement.

Another property required for processors is that they need to provide a *no-args* constructor, i.e. they need to have a constructor that comes with no arguments.

For a wide range of common preprocessing tasks, this simple method is sufficient enough to handle data. The processors might also maintain a state over consecutive calls to the

`process(Data)` method. This `process` method will be called from within a single thread only.

If a processor requires a more sophisticated configuration, e.g. for initializing a database connection at startup or release a file handle at shutdown, the `StatefulProcessor` interface can be used. In addition to the simple `Processor` interface, the stateful version adds two additional methods:

```
public interface StatefulProcessor extends Processor {
   /**
    *  Initialize data structures, open connections,...
    */
   public void init(ProcessContext ctx) throws Exception;

   /**
    *  Close connections, release resources,...
    */
   public void finish() throws Exception;
}
```

Figure 20: The additional methods of stateful processors.

5.1.1 The Lifecycle of a Processor

As stated above, a processor is expected to follow some basic conventions of the JavaBeans specification. It is expected to provide a constructor with no arguments and should provide access to attributes that are intended to be configurable via the XML configuration by providing `set`- and `get`-methods.

The general life-cycle of a processor that has been added to a data flow graph is as follows:

1. An object of the processor class is being instantiated at container startup time.
2. The parameters found as the XML attributes are used to call any `set`-methods that match the attribute names.
3. If the processor class implements the `StatefulProcessor` interface, the `init(ProcessContext)` method will be called.
4. The `process(Data)` method is called for all data items that the parent process of the processor receives.
5. As the container shuts down, the `finish()` method of the processor is called *if* the processor class implements the `StatefulProcessor` interface.

5.1.2 Example: A simple custom processor

In the following, we will walk through a very simple example to show the implementation of a processor in more detail. We will start with a basic class and extend this to have a complete processor in the end.

The main construct is a Java class within a package `my.package` that implements the identity function is given as:

```
package my.package;

public class Multiplier implements Processor {
     public Data process( Data item ){
         return item;
     }
}
```

This class implements a processor that simply passes through each data item to be further processed by all subsequent processors. Once compiled, this simple processor is ready to be used within a simple stream processing chain. To use it, we can directly use the XML syntax of the *streams* framework to include it in to the process:

```
<container>
  <process input="...">
       <!-- simply add an XML element for the new processor   -->
       <my.package.Multiplier />
  </process>
</container>
```

Figure 21: The processors are added to the XML process definition by simply adding an XML element with the name of the implementing class into the process that should contain the processor.

Processing data

The simple example shows the direct correspondence between the XML definition of a container and the associated Java implemented processors. The data items are represented as simple Hashmaps with `String` keys and `Serializable` values.

The code in Figure 22 extends the empty data processor from above by checking for the attribute with key `x` and adding a new attribute with key `y` by multiplying `x` by 2. This simple multiplier relies on parsing the double value from its string representation. If the double is available as Double object already in the item, then we could also directly cast the value into a Double:

```
// directly cast the serializable value to a Double object:
Double x = (Double) item.get( "x" );
```

The multiplier will be created at the startup of the experiment and will be called (i.e. the `process(..)` method) for each event of the data stream.

```
package my.package;
import stream.*;

public class Multiplier implements Processor {
    public Data process( Data item ){
        Serializable value = item.get( "x" );

        if( value != null  ){
           Double x = new Double( value.toString() );  // parse value to double
           data.put( "y",  new Double(  2 * x ) );     // multiply+add result
        }
        return item;
    }
}
```

Figure 22: A simple custom processor that multiplies an attribute `x` in each data item by a constant factor of 2. If the attribute `x` is not present, this processor will leave the data item unchanged.

5.1.3 Adding Parameters to Processors

In most cases, we want to add a simple method for parameterizing our Processor implementation. This can easily be done by following the *Convention-over-Configuration* paradigm: By convention, all `setX(...)` and `getY()` methods are automatically regarded as parameters for the data processors and directly available as XML attributes.

In the example from above, we want to add two parameters: `key` and `factor` to our Multiplier implementation. The `key` parameter will be used to select the attribute used instead of `x` and the `factor` will be a value used for multiplying (instead of the constant `2` as above).

To add these two parameters to our Multiplier, we only need to provide corresponding getters and setters as shown in Figure 24.

After compiling this class, we can directly use the new parameters `key` and `factor` as XML attributes. For example, to multiply all attributes `z` by `3.1415`, we can use the following XML setup:

```
<container>
    ...
    <process input="...">
        <my.package.Multiplier key="z" factor="3.1415" />
     </process>
</container>
```

Figure 23:

Upon startup, the getters and setters of the Multiplier class will be checked and if the argument is a Double (or Integer, Float,...) it will be automatically converted to that type.

In the example of our extended Multiplier, the `factor` parameter will be created to a Double object of value `3.1415` and used as argument in the `setFactor(..)` method.

```
// imports left out for truncation
//
public class Multiplier implements Processor {
   String key = "x";    // by default we still use 'x'
   Double factor = 2;   // by default we multiply with 2

   // getter/setter for parameter "key"
   //
   public void setKey( String key ){
       this.key = key;
   }

   public String getKey()(
       return key;
   }

   // getter/setter for parameter "factor"
   //
   public void setFactor( Double fact ){
       this.factor = fact;
   }

   public Double getFactor(){
       return factor;
   }
}
```

Figure 24: The `Multiplier` processor with added parameters.

5.2 Using Scripting Languages in *streams*

Scripting languages provide a convenient way to integrate ad-hoc functionality into stream processes. Based on the Java Scripting Engine that is provided within the Java virtual machine, the streams library includes support for several scripting languages, most notably the JavaScript language.

Additional scripting languages are being supported by the ScriptingEngine interfaces of the Java virtual machine. This requires the corresponding Java implementations (Java archives) to be available on the classpath when starting the *streams* runtime.

Currently the following scripting languages are supported:

- JavaScript (built into the Java VM)

- JRuby (requires jruby-library in classpath).

Further support for integrating additional languages like Python is planned.

5.2.1 Using JavaScript for Processing

The JavaScript language has been part of the Java API for some time. The *streams* framework provides a simple `JavaScript` processor, that can be used to run JavaScript functions on data items as shown in Figure 25.

```
<container>
   ...
   <process input="...">
       <!--  Execute a process(data) function defined in the
             specified JavaScript file     -->
       <JavaScript file="/path/to/myScript.js" />
   </process>
</container>
```

Figure 25: The `JavaScript` processor applies process() functions defined in JavaScript.

Within the JavaScript environment, the data items are accessible at `data`. Figure 26 shows an example for the JavaScript code which implements a processor within the file `myScript.js`.

```
function process(data){
   var id = data.get( "@id" );
   if( id != null ){
      println( "ID of item is: " + id );
   }
   return data;
 }
```

Figure 26: JavaScript code that implements a processor.

6 Example Applications

In this section we will give a more detailed walk-through of some applications and use-cases that the *streams* library is used for. These examples come from various domains, such as pre-processing of scientific data, log-file processing or online-learning by integrating the MOA library.

Most of the use-cases require additional classes for reading and processing streams, e.g. stream implementations for parsing domain-specific data formats. Due to the modularity of the *streams* library, domain-specific code can easily be added and directly used within the process design.

6.1 FACT Data Analysis

The first use-case we focus on is data pre-processing in the domain of scientific data obtained from a radio telescope. The FACT project maintains a telescope for recording cosmic showers in a fine grained resolution. The telescope consists of a mirror area which has a 1440-pixel camera mounted on top as shown in Figure 27. This camera is recording electric energy-impulses which in turn is measured by sampling each pixel at a rate of 2 GHz.

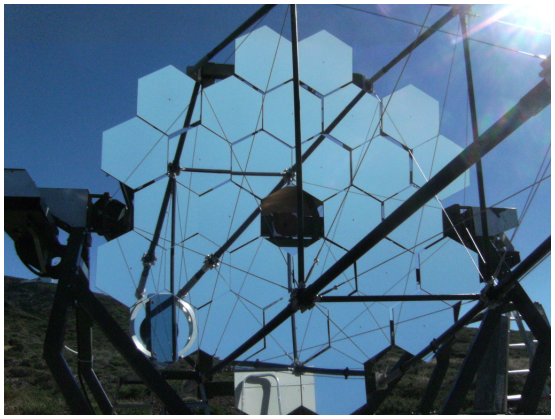


Figure 27: The FACT telescope on La Palma.

Based on trigger-signals, small sequences (a few nanoseconds) of these energy-impulses are recorded and stored into files. Each sequence is regarded as a single *event*. The electronics of the telescope are capable of recording about 60 events per second, resulting in a data volume of up to 10 GB of raw data that is recorded over a 5 minute runtime. The captured data of those 5-minute runs is stored in files.

The long-term objective of analyzing the FACT data comprises several tasks:

1. Identify events that represent showers.
2. Classify these events as Gamma or Hadron showers.
3. Use the Gamma events for further analysis with regard to physical analysis methods.

Besides the pure analysis the data has to be pre-processed in order to filter out noisy events, calibrate the data according to the state of the telescope electronic (e.g. stratify voltages over all camera pixels).

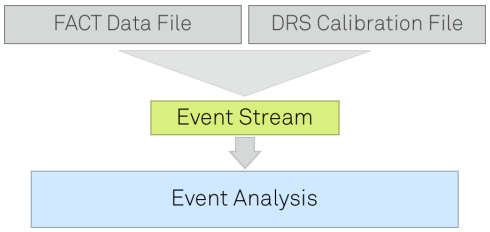


Figure 28: Stream-lined processing of events.

6.1.1 Reading FACT Data

The *fact-tools* library is an extension of the *streams* framework that adds domain specific implementations such as stream-sources and specific processors to process FACT data stored in FITS files. These processors provide the code for calibrating the data according to previously observed parameters, allow for camera image analysis (image cleaning) or for extracting features for subsequent analysis.

The XML snippet in Figure 29 defines a simple process to read raw data from a FITS file and apply a calibration step to transform that data into correct values based upon previously recorded calibration parameters.

```
<container>
    <stream id="factData" url="file:/data/2011-09-13-004.fits.gz"
         class="fact.io.FACTEventStream" />

    <process input="factData">
         <fact.io.DrsCalibration file="/data/2011-09-13-001.fits.drs.gz" />
         <!--  add further processors here  -->
    </process>
</container>
```

Figure 29: Basic process definition for reading raw FACT data and calibrating that data.

Each single event that is read from the event stream, contains the full raw, calibrated measurements of the telescope. The attributes of the data items reflect the image data, event meta information and all other data that has been recorded during the observation. Table 3 lists all the attributes of an event that are currently provided by the `FACTEventStream` class.

The `@id` and `@source` attributes provide meta-information that is added by the FACT-stream implementation itself, all the other attributes are provided within the FITS data files. The `@id` attribute's value is created from the `EventNum` and date when the event was recorded, e.g. `2011/11/27/42/8`, denoting the event 8 in run 42 on the 27th of November 2011.

Name (key)	Description
`EventNum`	The event number in the stream
`TriggerNum`	The trigger number in the stream
`TriggerType`	The trigger type that caused recording of the event
`NumBoards`	
`Errors`	
`SoftTrig`	
`UnixTimeUTC`	
`BoardTime`	
`StartCellData`	
`StartCellTimeMarker`	
`Data`	The raw data array ($1440 \cdot 300 = 432000$ float values)
`TimeMarker`	
`@id`	A simple identifier providing date, run and event IDs
`@source`	The file or URL the event has been read from

Table 3: The elements available for each event.

6.1.2 Processors for FACT Data

Any of the existing core processors of the *streams* library can directly be applied to data items of the FACT event stream. This already allows for general applications such as adding additional data (e.g. whether data from a database).

The *fact-tools* library provides several domain specific processors that focus on the handling of FACT events. The `DrsCalibration` processor for calibrating the raw data has already been mentioned above.

Other processors included are more specifically addressing the image-analysis task:

- `fact.data.CutSlices`
  Which can be used to select a subset of the raw data array for only a excerpt of the region-of-interest[3] (ROI).
- `fact.data.SliceNormalization`
  As there is a single-valued series of floats provided for each pixel, this processor allows for normalizing the values for these series to $[0, 1]$.
- `fact.data.MaxAmplitude`
  This processor extracts a float-array of length 1440, which contains the maximum amplitude for each pixel.
- `fact.image.DetectCorePixel`
  This class implements a heuristic strategy to select the possible core-pixels of a shower, that may be contained within the event.

[3]The *region of interest* is the length of the recorded time for each event, usually 300 nanoseconds, at most 1024 nanoseconds

7 Summary and Future Work

In this report we introduced the *streams* framework, which provides means for abstracting the definition of data flow graphs for data stream processing. The level of abstraction provided by the *streams* framework enables a rapid prototyping of compute graphs for process design as well as providing a simple programming API to include custom functionality into the designed processes.

The use of XML for process/graph definitions supports a simple exchange of designed processes between users and lifts the level of detail for data analysists to hide implementation details where they may be distracting from the process design task.

The *streams* framework also provides a reference implementation for running compute graphs on a single Java virtual machine as well as a compiler for mapping graphs to topologies that execute on the *Storm* stream engine.

The integration of the *MOA* library adds various online learning schemes to the *streams* framework. This shows the applicability of the proposed abstraction layer for the field of online learning. In addition the *streams* library proved to be useful in application use-cases like pre-processing of the FACT telescope data or the coffee machine video processing.

Ongoing work currently focuses on a more extensive integration of additional algorithms provided by *MOA* (e.g. clustering). The adaption of the *streams* runtime for the Android platform has revealed a prototype for running XML process definitions on mobile devices. This is another direction that will be integrated into the next release of the *streams* framework.

For a more convenient design of *streams* process definitions, we will investigate different XML or process editors that can assist users in rapid prototyping of data stream processes.

Part II

API Documentation

The intention of the *streams* framework is to start from a minimal set of core concepts and allow for building solutions for more complex problems on top of that. The main part of this report is a description of the concepts that mainly focus on establishing a data-flow definition language.

For running data flow experiments defined with the *streams* we provide an introduction into the *streams runtime* environment in Section A.

The remainder of this document includes a comprehensive documentation of a set of processors that constitute the *streams* API. These elements provide a toolbox for handling various processing steps and are described in Section B.

A The *streams* Runtime

Along with the *streams* API, that is provided for implementing custom streams or processors, the *streams* framework provides a runtime environment for running stream containers.

A.1 Running Streams - Quickstart

Designing a simple stream process does not require more than writing some XML declaration and executing that XML with the stream-runner as shown in the following figure:

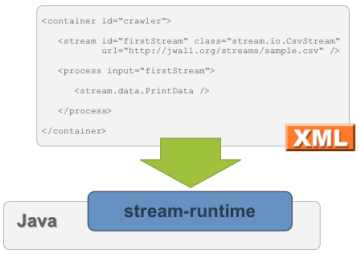


Figure 30: Conceptual way of executing a data flow graph that is defined in XML.

The simple example presented below, defines a single process that reads from a CSV stream and prints out the data items to standard output:

```
<container>
    <stream id="firstStream" class="stream.io.CsvStream"
            url="http://www.jwall.org/streams/sample-stream.csv" />

    <process input="firstStream">
        <PrintData />
    </process>
 </container>
```

The *stream-runner* required to execute this stream is a simple executable Java archive available for download:

`http://download.jwall.org/streams/stream-runner.jar`

A.1.1 Running a *streams* Process

The simple process defined above can be run by

```
# java -jar stream-runner.jar first-process.xml
```

The process will simply read the stream in CSV-format and execute the processor `PrintData` for each item obtained from the stream.

A.1.2 Including additional Libraries

There exists a set pre-packaged libraries such as *streams-analysis* or *streams-video*, which are provided at

`http://download.jwall.org/streams/libs/`

These add additional processors and stream implementations to the *streams* runtime for different domain specific intentions. To start the *streams* runtime with additional libraries, these need to be provided on the classpath.

The following example uses the `MJpegImageStream` to process a stream of video data from some URL. This stream implementation is provided in the *streams-video* package.

```
<container>
    <stream id="video"
         class="stream.io.MJpegImageStream"
           url="http://download.jwall.org/streams/coffee.mjpeg.gz" />

    <process input="video" >

        <stream.image.DisplayImage key="data" />
        <stream.image.AverageRGB />

        <WithKeys keys="frame:*">
             <stream.plotter.Plotter
                     history="1000"
                    keepOpen="true"
                    keys="frame:red:avg,frame:green:avg,frame:blue:avg" />
        </WithKeys>
    </process>
</container>
```

Figure 31: Displaying an MJPEG video stream and plotting the average RGB channels.

For the libraries to be included in the path, the following command needs to be issued to start the *streams* run-time:

```
# java -cp stream-runner.jar:streams-video-0.0.1.jar \
stream.run video.xml
```

A.2 Installing the *streams* Runtime package

For a more convenient use of the *streams* framework, we created packages for the major Linux systems, such as Debian and RedHat based systems. These packages install into a proper directory structure and allow for a convenient use of *streams* .

The packages provide an `streams.run` command that automatically uses all libraries found in `/opt/streams/lib`. With the *streams* package installed, any container XML file can be started by running

```
# streams.run my-container.xml
```

To add additional libraries or custom processors, these simply need to be packaged into a Jar file and added to the `/opt/streams/lib` directory.

The following sections will give a more detailed instructions on how to install the *streams* packages on the different platforms.

A.2.1 Installing the *streams* Runtime on Debian/RedHat

For Debian and RPM based systems, there exists a package repository, that provides Debian and RPM packages that can easily be installed using the system's package managers. A step-by-step guide for setting up the package manager on Debian and Ubuntu systems is provided in Section A.2.1. Instructions for RedHat based systems such as RedHat, CentOS or Scientific Linux are provided in A.2.2.

Signatures for Packages

The repositories and all packages within the repository are cryptographically signed with a GPG key with ID `0x13443F4A` to ensure their consistency. The key is available at

```
http://download.jwall.org/software.gpg
```

The key is associated with the following information:

```
User ID: Christian Bockermann <chris@jwall.org>
Fingerprint: 4324 5FA1 EA37 1C3E EFE3 0730 A5CE 7F45 C5C3 953C
```

This key needs to be added to the package management key ring of the system (e.g. *apt* on Debian or *yum* on RedHat systems).

Installing *streams* on Debian/Ubuntu

There exists a Debian/Ubuntu repository at `jwall.org`[4] which provides access to the latest release versions of the *streams* library.

To access this repository from within your Debian system, you'll need create a new file `/etc/apt/sources.list.d/jwall.list` with the following content:

```
deb http://download.jwall.org/debian/ jwall main
```

The repositories and all packages within the repository are cryptographically signed with a GPG key. Please see Section A.2.1 above for details on how to verify the correctness of this key.

This key needs to be added to the APT key ring of the Debian/Ubuntu system by running the following commands (the `#` denotes the shell prompt):

[4]The site `http://www.jwall.org/streams/` is the base web-site of the *streams* framework.

```
# sudo wget http://download.jwall.org/debian/software.gpg
# sudo apt-key add software.gpg
```

After the key and the repository have been added to the APT package management, all that is left is to update the package list and install the *streams* environment with the following commands:

```
# sudo apt-get update
# sudo apt-get install streams
```

The first command will update the package lists, the second will install the lastest version of the `streams` package. After installation, the system should be equipped with a new `stream.run` command to run XML stream processes:

```
# stream.run my-process.xml
```

A.2.2 Installing *streams* on RedHat/CentOS/Fedora

There exists a YUM repository at the `jwall.org` site, which provides access to the latest release versions of the *streams* framework for RedHat based systems.

To access this repository from within your CentOS/RedHat system, you'll need to create a file `/etc/yum.repos.d/jwall.repo` with the following contents:

```
[jwall]
name=CentOS-jwall - jwall.org packages for noarch
baseurl=http://download.jwall.org/yum/jwall
enabled=1
gpgcheck=1
protect=1
```

The RPM packages are signed with a GPG key, please see Section A.2.1 for information how to validate this key.

To import the GPG key into your system's key ring, run the following command as super user:

```
# rpm -import http://download.jwall.org/software.gpg
```

After the key has been imported your system is ready to install the *streams* package using the system's package manager, e.g. by running

```
# yum install streams
```

This will download the required packages and set up the system to provide the `stream.run` command to execute XML stream processes.

B The *streams* Core Classes

The *streams* framework provides a wide range of implementations for data streams and processors. These are useful for reading application data and defining a complete data flow.

In this section we provide a comprehensive overview of the classes and implementations already available in the *streams* library. These can directly be used to design stream processes for various application domains.

B.1 Data Stream Implementations

Reading data is usually the first step in data processing. The package `stream.io` provides a set of data stream implementations for data files/resources in various formats.

All of the streams provided by this package do read from URLs, which allows reading from files as well as from network URLs such as HTTP urls or plain input streams (e.g. standard input).

The streams provide an iterative access to the data and use the default `DataFactory` for creating data. They do usually share some common parameters supported by most of the streams such as `limit` or `username` and `password`.

Defining a Stream

As discussed in Section 3, a stream is defined within a container using the XML `stream` element, providing a `url` and `class` attribute which determines the source to read from and the class that should be used for reading from that source. In addition, the definition requires a third attribute `id`, which assigns the stream with a (unique) identifier. This identifier is then used to reference the stream as input to a process.

As a simple example, the following XML snippet defines a data stream that reads data items in CSV format from some file URL:

```
<stream  id="csv-data" class="stream.io.CsvStream"
        url="file:/tmp/example.csv" />
```

Figure 32: Defining a CSV stream from a file.

Streaming Data from various URLs

The *streams* runtime supports a list of different URL schemes which are provided by all Java virtual machines, e.g. `http` URLs or `file` URLs. Custom URL schemes can also be registered within the Java VM. As of this, the *streams* runtime additionally offers a `classpath:` and a `system:` URL scheme.

The `classpath:` URLs can be used to create data streams that read from resources which are available on the classpath. This is useful for providing example sources within custom JAR files or the like. The following example shows how to create a stream that reads data in JSON format from a resource `example.json` that is searched for in the default classpath:

```
<stream  id="json-stream"  class="stream.io.JSONStream"
        url="classpath:/example.json" />
```

Figure 33: Defining a JSON stream from a classpath resource.

To support streams that read data from standard input or standard error, the library provides the `system:` URL schema. This schema provides access to the system input and error streams and are useful when piping data to a stream via the command line, e.g. by running a command like: To define a stream that reads from standard input, simply

```
# cat data.csv | stream.run my-process.xml
```

specify `system:input` as the streams URL as shown in figure

```
<stream  id="example"  class="stream.io.CsvStream"
        url="system:input" />
```

Figure 34: Defining a CSV stream that reads data from the system's standard input.

B.1.1 ArffStream

This stream implementation provides access to reading ARFF files and processing them in a stream based fashion. ARFF is a standard format for data in the machine learning community which has its root in the WEKA project [13].

Parameter	Type	Description	Required
`id`	String	The identifier to reference this stream in the container	true
`password`	String	The password for the stream URL (see username parameter)	false
`prefix`	String	An optional prefix string to prepend to all attribute names	false
`limit`	Long	The maximum number of items that this stream should deliver	false
`username`	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Table 4: Parameters of class `stream.io.ArffStream`

B.1.2 CsvStream

This data stream source reads simple comma separated values from a file/url. Each line is split using a separator (regular expression).

Lines starting with a hash character (`#`) are regarded to be headers which define the names of the columns.

The default split expression is `(;|,)`, but this can changed to whatever is required using the `separator` parameter.

Parameter	Type	Description	Required
`keys`	String[]		?
`separator`	String		true
`id`	String		?
`password`	String	The password for the stream URL (see username parameter)	false
`prefix`	String	An optional prefix string to prepend to all attribute names	false
`limit`	Long	The maximum number of items that this stream should deliver	false
`username`	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Figure 35: Parameters of class `stream.io.CsvStream`.

B.1.3 JSONStream

This data stream reads JSON objects from the source (file/url) and returns the corresponding Data items. The stream implementation expects each line of the file/url to provide a single object in JSON format.

B.1.4 LineStream

This class provides a very flexible stream implementations that essentially reads from a URL line-by-line. The content of the complete line is stored in the attribute determined by the `key` parameter. By default the key `LINE` is used.

It also supports the specification of a simple format/grammar string that can be used to create a generic parser to populate additional fields of the data item read from the stream.

The grammar is a string containing `%(name)` elements, where `name` is the name of the attribute that should be created at that specific portion of the line. An example, for such a simple grammar is given as follows:

```
%(IP) [%(DATE)] "%(URL)"
```

The `%(name)` elements are extracted from the grammar and all remaining elements in between are regarded as boundary strings that separate the elements.

The simple grammar above will create a parser that is able to read lines in the format of the following:

```
127.0.0.1 [2012/03/14 12:03:48 +0100] "http://example.com/index.html"
```

The outcoming data item will have four attributes `LINE`, `IP`, `DATE` and `URL`. The attribute `IP` set to `127.0.0.1` and the `DATE` attribute set to `2012/03/14 12:03:48 +0100`. The `URL` attribute will be set to `http://example.com/index.html`. The `LINE` attribute will contain the complete line string.

Parameter	Type	Description	Required
`id`	String	The ID of the stream with which it is assicated to proceses.	true
`key`	String	The name of the attribute holding the complete line, defaults to `LINE`.	false
`format`	String	The format how to parse each line. Elements like `%(KEY)` will be detected and automatically populated in the resulting items.	false
`password`	String	The password for the stream URL (see username parameter)	false
`prefix`	String	An optional prefix string to prepend to all attribute names	false
`limit`	Long	The maximum number of items that this stream should deliver	false
`username`	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Table 5: Parameters of class `stream.io.LineStream`

B.1.5 ProcessStream

This processor executes an external process (programm/script) that produces data and writes that data to standard output. This can be used to use external programs that can read files and stream those files in any of the formats provided by the stream API.

The default format for external processes is expected to be CSV. In the following example, the Unix command `cat` is used as an example, producing lines of some CSV file:

```
<stream   class="stream.io.ProcessStream"
        command="/bin/cat␣/tmp/test.csv"
         format="stream.io.CsvStream" />
```

The process is started at initialization time and the output will be read from standard input.

Parameter	Type	Description	Required
`id`	String	The ID of the stream with which it is assicated to proceses.	true
`format`	String	The format of the input (standard input), defaults to CSV	true
`command`	String	The command to execute. This command will be spawned and is assumed to output data to standard output.	true

Table 6: Parameters of class `stream.io.ProcessStream`.

B.1.6 SQLStream

This class implements a DataStream that reads items from a SQL database table. The class requires a `jdbc` URL string, a username and password as well as a `select` parameter that will select the data from the database.

The following XML snippet demonstrates the definition of a SQL stream from a database table called `TEST_TABLE`: The database connection is established using the user `SA` and

```
<stream class="stream.io.SQLStream"
          url="jdbc:mysql://localhost:3306/TestDB"
     username="SA" password=""
       select="SELECT␣*␣FROM␣TEST_TABLE" />
```

Figure 36: Example SQL streams, reading from a database.

no password (empty string). The above example connects to a MySQL database.

As the SQL database drivers are not part of the streams library, you will need to provide the database driver library for your database on the class path.

Parameter	Type	Description	Required
`id`	String	The ID of the stream with which it is assicated to proceses.	true
`url`	String	The JDBC database url to connect to.	true
`select`	String	The select statement to select items from the database.	true
`password`	String	The password for the stream URL (see username parameter)	false
`prefix`	String	An optional prefix string to prepend to all attribute names.	false
`limit`	Long	The maximum number of items that this stream should deliver.	false
`username`	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Table 7: Parameters of class `stream.io.SQLStream`.

B.1.7 SvmLightStream

This stream implementation provides a data stream for the SVMlight format. The SVMlight format is a simple `key:value` format for compact storage of high dimensional sparse labeled data. It is a line oriented format where each line is laid out as shown in Figure 37. The keys are usually indexes, but this stream implementation also supports string keys. The # character starts a comment that can be provided to each line.

```
-1.0 4:3.3 10:0.342 44:9.834 # some comment
```

Figure 37: A sample line of a SVMLight file.

Parameter	Type	Description	Required
`sparseKey`	String		?
`id`	String	The ID of this string for associating it with processes.	true
`password`	String	The password for the stream URL (see username parameter)	false
`prefix`	String	An optional prefix string to prepend to all attribute names.	false
`limit`	Long	The maximum number of items that this stream should deliver.	false
`username`	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Table 8: Parameters of class `stream.io.SvmLightStream`.

B.1.8 TimeStream

This is a very simple stream that emits a single data item upon every read. The data item contains a single attribute `@timestamp` that contains the current timestamp (time in milliseconds).

The name of the attribute can be changed with the `key` parameter, e.g. to obtain the timestamp in attribute `@clock`:

```
<Stream class="stream.io.TimeStream" key="@clock" />
```

Parameter	Type	Description	Required
`id`	String	The ID of this string for associating it with processes.	true
`key`	String	The name of the attribute that should hold the timestamp, defaults to `@timestamp`	false
`interval`	String	The time gap/rate at which this stream should provide items.	true
`prefix`	String	An optional prefix string to prepend to all attribute names.	false
`limit`	Long	The maximum number of items that this stream should deliver.	false

Table 9: Parameters of class `stream.io.TimeStream`.

B.2 Queue Implementations

The notion of queues is similar to the definition of streams within the *streams* framework. Queues provide can be attached as sources to processes while also allowing to be fed with data items from other places. This allows for simple inter-process communication by forwarding data items from one process to the queue that is read by another different process.

B.2.1 BlockingQueue

The class `stream.io.BlockingQueue` provides a simple DataStream that items can be enqueued into and read from. This allows inter-process communication between multiple active processes to be designed using data items as messages.

As the name already suggests, this queue is a blocking queue, resulting in any process that reads from this queue to block if the queue is empty. Likewise, any processor that adds items to the queue (e.g. `stream.flow.Enqueue`) will be blocking if the queue is full.

By default the size of the queue is unbounded (i.e. bound by the available memory only), but can be fixed by using the `size` parameter.

Parameter	Type	Description	Required
`size`	Integer	The maximum number of elements that can be held in the queue.	
`id`	String	The ID of this queue for associating it with processes.	true
`password`	String	The password for the stream URL (see username parameter)	false
`prefix`	String	An optional prefix string to prepend to all attribute names.	false
`limit`	Long	The maximum number of items that this stream should deliver.	false
`username`	String	The username required to connect to the stream URL (e.g web-user, database user)	false

Table 10: Parameters of class `stream.io.BlockingQueue`.

B.3 The *stream-core* Processor Classes

The core packages of the *streams* framework provide a set of processor implementations that cover a lot of general stream processing tasks. These processors serve as basic building blocks to design a stream process. A processor can simply be added to a process by adding an XML element with the name of the processor to the process element as shown in Figure 38.

Based on their purpose, the processors are organized into packages. To shorten the XML declaration, several packages are automatically checked for when resolve a processor. For

```
<process input="my-stream">
    <!--  convert the string of attribute x1 to a double value -->
    <stream.parser.ParseDouble key="x1" />
</process>
```

Figure 38: The processor `stream.parser.ParseDouble` added to a process.

example the `stream.parser` package is among the default base packages. This allows for leaving out the package name, when adding the processor. Thus, the XML from Figure 38 is equivalent to the following XML snippet:

```
<process input="my-stream">
    <!--  convert the string of attribute x1 to a double value -->
    <ParseDouble key="x1" />
</process>
```

The default packages that are automatically checked for when resolving processor names are:

- `stream.data`
- `stream.flow`
- `stream.parser`
- `stream.script`.

B.3.1 Processors in Package `stream.flow`

The `stream.flow` package contains processors that allow for data flow control within a process setup. Processors in this package are usually processor-lists, i.e. they may provide nested processors that are executed based on conditions.

A typical example for control flow is given with the following `If` processor, which executes the `PrintData` processor only, if the value of attribute `x1` is larger than 0.5. Other flow

```
<If condition="\%{data.x1}␣@gt␣0.5">
  <PrintData />
</If>
```

control processors provide control of data queues such as enqueuing events into other processes' queues.

B.3.1.1 Processor `Delay`

This simple processor puts a delay into the data processing. The delay can be specified in various units with the simple time format being specified like "`40ms`" for specifying a delay of 40 milliseconds. Other units for `day`, `hour`, `minute` and so on work as well.

The units can also be combined as in `1 second 30ms`.

Parameter	Type	Description	Required
`time`	String	The time that the data flow should be delayed.	true
`condition`	String	The condition parameter allows to specify a boolean expression that is matched against each item. The processor only processes items matching that expression.	false

Table 11: Parameters of class `stream.flow.Delay`.

B.3.1.2 Processor `Enqueue`

This processor will enqueue data items into specified queues. To ensure mutual access to the data, the items are copied and copies are sent to the queues. This may lead to a multiplication of data.

The processor is a conditioned processor, i.e. it supports the use of condition expressions. As an example, the XML snippet in Figure 39 will enqueue all events with a `color` value equal to `blue` into the queue `blue-items`.

```
<process ...>
    <Enqueue queues="blue-items" condition="%{data.color} == blue" />
</process>
```

Figure 39: The `Enqueue` processor combined with a condition.

Parameter	Type	Description	Required
`queues`	ServiceRef[]	A list of names that reference the target queues.	true
`condition`	Condition	A condition that is required to evaluate to *true* for this processor to be executed. If no condition is specified, then the processor is executed for every data item.	false

Table 12: Parameters of class `stream.data.Enqueue`.

B.3.1.3 Processor `Every`

This processor requires a parameter `n` and will then execute all inner processors every `n` data items, i.e. if the number of observed items modulo `n` equals 0.

In all other cases, the inner processors will simply be skipped.

Parameter	Type	Description	Required
`n`	Long		?

Table 13: Parameters of class `stream.flow.Every`.

B.3.1.4 Processor `If`

This processor provides conditioned execution of nested processors. By specifying a condition, all nested processors are only executed if that condition is fulfilled.

As an example, the following will only print data items if the attribute `x` is larger than `3.1415`:

```
<If condition="%{data.x} @gt 3.1415">
   <PrintData />
</If>
```

Parameter	Type	Description	Required
`condition`	String		false

Table 14: Parameters of class `stream.flow.If`.

B.3.1.5 Processor `OnChange`

The *OnChange* processor is a processor list that executes all nested processors if some state has changed. This is similar to the *If* processor, but provides support for a process state check.

In the following example, the *Message* processor is only executed if the context variable `status` changes from `green` to `yellow`:

```
...
<OnChange from=‘green‘ to=‘yellow‘>
   <Message message="Status change detected!" />
</OnChange>
```

Parameter	Type	Description	Required
key	String		true
from	String		false
to	String		false
condition	String		false

Table 15: Parameters of class `stream.flow.OnChange`.

B.3.1.6 Processor `Skip`

This processor will simply skip all events matching a given condition. If no condition is specified, the processor will skip all events.

The condition must be a bool expression created from numerical operators like `@eq`, `@gt`, `@ge`, `@lt` or `@le`. In addition to those numerical tests the `@rx` operator followed by a regular expression can be used.

The general syntax is

```
variable  operator  argument
```

For example, the following expression will check the value of attribute `x1` against the 0.5 threshold:

```
%{data.x1} @gt 0.5
```

Parameter	Type	Description	Required
condition	String	The condition parameter allows to specify a boolean expression that is matched against each item. The processor only processes items matching that expression.	false

Table 16: Parameters of class `stream.flow.Skip`.

B.3.1.7 Processor `Collect`

This processor requires a `count` parameter and a `key` to be specified. The implementation will wait for a number of `count` data items and collect these in a list. As soon as `count` items have been collected, a new, empty item will be created which holds an array of the collected items in the attribute specified by `key`.

While waiting for `count` items to arrive, the processor will return `null` for each collected data item, such that no subsequent processors will be executed in a process.

After emitting the collected data items, the counter is reset and the processor starts collecting the next `count` items.

Parameter	Type	Description	Required
`key`	String	The key (name) of the attribute into which the collection (array) of items will be put, defaults to '@items'	false
`count`	Integer	The number of items that should be collected before the processing continues.	true

Table 17: Parameters of class `stream.flow.Collect`.

B.3.1.8 Processor `ForEach`

This class implements a processor list. It can be used if the current data item provides an attribute that holds a collection (list, set, array) of data items, which need to be processed.

The `ForEach` class extracts the nested collection of data items and applies each of the inner processors to each data item found in the collection. The `key` parameter needs to be specified to define the attribute which holds the collection of items.

If no key is specified or the data item itself does not provide a collection of items in this key, then this processor will simply return the current data item.

Parameter	Type	Description	Required
`key`	String	The name of the attribute containing the collection of items that should be processed.	false

Table 18: Parameters of class `stream.flow.ForEach`.

B.3.2 Processors in Package `stream.data`

This package provides processors that perform transformations or mangling of the data items themselves. Examples for such processors are `CreateID`, which adds a sequential ID attribute to each processed item or the `RemoveKeys` processor which removes attributes by name.

Other useful processors provide numerical binning (`NumericalBinning`), setting of values in various scopes (`SetValue`) and the like.

B.3.2.1 Processor `AddTimestamp`

This processor simply adds the current time as a UNIX timestamp to the current data item. The default attribute/key to add is `@timestamp`.

The value is the number of milliseconds since the epoch date, usually 1.1.1970. Using the `key` parameter, the name of the attribute to add can be changed:

```
<stream.data.AddTimestamp key="@current-time" />
```

Parameter	Type	Description	Required
key	String	The key of the timestamp attribute to add	false

Table 19: Parameters of class `stream.data.AddTimestamp`.

B.3.2.2 Processor `WithKeys`

This processor is a processor list that executes one or more inner processors. It creates a copy of the current data item with all attributes matching the list of specified keys. Then all nested processors are applied to that copy and the copy is merged back into the original data item.

If any of the nested data items returns *null*, this processor will also return *null*.

The `keys` parameter of this processor allows for specifying a comma separated list of keys and key-patterns using simple wildcards `*` and `?` as shown in Figure 40. If the `keys` parameter is not provided, then the inner processors will be provided with a complete copy of the current data item.

```
<process ...>
    <WithKeys keys="x1,user:*,!user:id">
       <PrintData />
    </WithKeys>
</process>
```

Figure 40: Selects only attribute `x1`, all attributes starting with `user:` but not attribute `user:id` and executes the `PrintData` processor for this selection of attributes.

Parameter	Type	Description	Required
keys	String[]	A list of filter keys selecting the attributes that should be provided to the inner processors.	false
merge	Boolean	Indicates whether the outcome of the inner processors should be merged into the input data item, defaults to true.	false

Table 20: Parameters of class `stream.data.WithKeys`.

B.3.2.3 Processor `SetValue`

This processors allows for setting a feature to a single, constant value:

```
<SetValue key="attribute1" value="abc" />
```

Parameter	Type	Description	Required
value	String		?
key	String	The name of the attribute to set.	true
scope	String[]	The scope determines where the variable will be set. Valid scopes are process, data. The default scope is data.	false
condition	String	The condition parameter allows to specify a boolean expression that is matched against each item. The processor only processes items matching that expression.	false

Table 21: Parameters of class stream.data.SetValue.

B.3.3 Processors in Package stream.parser

When processing streams of data each single data item may contain additional information that needs to be extracted into more detailed attributes or into other value types.

The stream.parser package provides a set of parsing processors, that usually act upon on or more keys and extract information from the attributes denoted by those keys.

For example the ParseDouble processor will parse double values from all strings that are denoted in its keys parameter. Other parsers in this package are for example the ParseJSON, Timestamp or the NGrams processor.

B.3.3.1 Processor NGrams

This parser processor will create n-grams from a specified attribute of the processed item and will add all the n-grams and their frequency to the item. By default the processor creates n-grams of length 3.

To not overwrite any existing keys, the n-gram frequencies can be prefixed with a user-defined string using the prefix parameter.

The following example shows an *NGram* processor that will create 5-grams of the string found in key text and add their frequency to the items with a prefix of 5gram:

```
<NGrams n="5" key="text" prefix="5gram" />
```

Parameter	Type	Description	Required
key	String	The attribute which is to be split into n-grams	true
n	Integer	The length of the n-grams that are to be created	true
prefix	String	An optional prefix that is to be prepended for all n-gram names before these are added to the data item	false

Table 22: Parameters of class stream.parser.NGrams.

B.3.3.2 Processor `ParseDouble`

This simple processor parses all specified keys into double values. If a key cannot be parsed to a double it will be replaced by *Double.NaN*.

The processor will be applied for all keys of an item unless the `keys` parameter is used to specify the keys/attributes that should be transformed into double values.

The following example shows a *ParseDouble* processor that converts the attributes `x1` and `x2` into double values:

```
<stream.parser.ParseDouble keys="x1,x2" />
```

The `default` parameter allows for specifying a different value than the default *Double.NaN* value. The following example converts all values to their double representation and defaults to 0.0 if parsing as double fails:

```
<stream.parser.ParseDouble default="0.0" />
```

Parameter	Type	Description	Required
`default`	Double	The default value to set if parsing fails	false
`keys`	String[]	The keys/attributes to perform parsing on	true

Table 23: Parameters of class `stream.parser.ParseDouble`.

B.3.3.3 Processor `ParseTimestamp`

This processor parses the date time from an attribute using a specified format string and stores the parsed time as a long value into the `@timestamp` key by default.

The processor requires at least a `format` and a `from` parameter. The `format` specifies a date format to parse the time from. The `from` parameter determines the key attribute from which the date is to be parsed.

The following example shows a timestamp parser that parses the `DATE` key using the format `yyyy-MM-dd-hh:mm:ss`. The resulting timestamp (milliseconds UNIX time) is stored under key `@time`:

```
<stream.parser.ParseTimestamp  key="@time" format="yyyy-MM-dd-hh:mm:ss"
                               from="DATE" />
```

B.3.4 Processors in Package `stream.script`

B.3.4.1 Processor `JRuby`

This processor executes JRuby (Ruby) scripts using the Java ScriptingEngine interface. To use this processor, the JRuby implementation needs to be available in the classpath.

Parameter	Type	Description	Required
key	String		false
format	String	The date format string used for parsing.	true
from	String	The key/attribute from which the timestamp should be parsed.	true
timezone	String	The timezone that the processed data is assumed to refer to.	false

Table 24: Parameters of class `stream.parser.ParseTimestamp`.

The script is evaluated for each processed item and will be provided to the script as variable `$data`.

Parameter	Type	Description	Required
file	File		false
script	BodyContent		false

Table 25: Parameters of class `stream.script.JRuby`.

B.3.5 JavaScript

This processor can be used to execute simple JavaScript snippets using the Java-6 ECMA scripting engine.

The processor binds the data item as `data` object to the script context to allow for accessing the item. The following snippet prints out the message "Test" and stores the string `test` with key `@tag` in the data object:

```
println( "Test" );
data.put( "@tag", "Test" );
```

B.3.5.1 External Scripts

The processor can also be used to run JavaScript snippets from external files, by simply specifying the `file` attribute:

```
<JavaScript file="/path/to/script.js" />
```

Parameter	Type	Description	Required
`file`	File		false
`script`	BodyContent		false

Table 26: Parameters of class `stream.script.JavaScript`.

B.3.6 Processors in Package `stream.image`

B.3.6.1 Processor `stream.image.AverageRGB`

This processor extracts RGB colors from a given image and computes the average RGB values over all pixels of that image.

References

[1] Marcel R. Ackermann, Christiane Lammersen, Marcus Märtens, Christoph Raupach, Christian Sohler, and Kamil Swierkot. Streamkm++: A clustering algorithms for data streams. In Guy E. Blelloch and Dan Halperin, editors, *ALENEX*, pages 173–187. SIAM, 2010.

[2] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 81–92. VLDB Endowment, 2003.

[3] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *PODS '04: Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 286–296, New York, NY, USA, 2004. ACM.

[4] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa massive online analysis, 2010. `http://mloss.org/software/view/258/`.

[5] Christian Bockermann and Hendrik Blom. Processing data streams with the rapidminer streams plugin. In *RCOMM 2012: RapidMiner Community Meeting And Conference*. Rapid-I, 2012.

[6] Toon Calders, Nele Dexters, and Bart Goethals. Mining frequent itemsets in a stream. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining*, ICDM '07, pages 83–92, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Moses Charikar, Kevin Chen, and Martin Farach-colton. Finding frequent items in data streams. pages 693–703, 2002.

[8] James Cheng, Yiping Ke, and Wilfred Ng. Maintaining frequent itemsets over high-speed data streams. In *In Proc. of PAKDD*, 2006.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[10] Pedro Domingos and Geoff Hulten. Mining High Speed Data Streams. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '00)*, pages 71–80, New York, NY, USA, 2000. ACM.

[11] Nathan Marz et.al. Twitter storm framework, 2011. `https://github.com/nathanmarz/storm/`.

[12] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. In *In SIGMOD*, pages 58–66, 2001.

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[14] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed Stream Computing Platform. In *Data Mining Workshops, International Conference on*, pages 170–177, CA, USA, 2010. IEEE Computer Society.

[15] H. G. Wells. *Pattern-orientierte Software-Architektur*. Addison Wesley Verlag, 1998.