The streams Library for Processing Streaming Data

Technical Report

Christian Bockermann
Lehrstuhl für künstliche Intelligenz
Technische Universität Dortmund
christian.bockermann@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

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 which follow the semantics and ideas of well established tools like Maven or the Spring Framework.

The *streams* library can be easily embedded into existing software, used as a standalone tool or be integrated into the RapidMiner tool using the *Streams Plugin*.

1 Introduction

In todays 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 poses tight challenges on the analysis of such streams.

Several projects within the Collaborative Research Center SFB-876 deal with data of large volume. An example is given by the FACT telescope that is associated to project C3. This telscope 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 of the oven 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.

1.1 From Batches to Streams

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.

To catch up with the reqirements 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. The constraints for the data stream setting are generally defined by allowing only a single pass over the data, and focusing on approximation schemes to deal with the inbalance of data volume to computing resources. In addition, models computed on streaming data are expected to be queriable at any time.

Various algorithms have been proposed dedicated to computational problems on data streams. Examples include online quantile computation [11, 3], distinct counting of elements, frequent itemset mining [6, 5, 7], clustering [1, 2] or training of classifiers on a stream [9].

1.2 Designing Stream Processes

In this work we introduce the *streams* library, a small software framework that provides an abstract modelling 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 [12] 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 runtime environments like *Storm* or the RapidMiner platform [**?**].

Our proposed framework supports

1. Modelling 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 modelling
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 review the problem setting and give an overview of related work and existing frameworks. Based on this we derive

some basic building blocks for a modeling data stream processes (Section 3). In Section **??** we present the *streams* API which provides implementations to these building blocks. Finally we summarize the ideas behind the *streams* library and give an outlook on future work.

2 Problem and Related Work

The traditional batch data processing aims at computations on fixed chunks of data in one or more passes. 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 training set. 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 aspects have been changed in the data we are facing today, which requires 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.

2.1 Processing Masses of Data

The former problem has 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 [8]. 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.

2.2 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 been 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 $i \to \infty$. At any time t we want to provide some model that reflects the analysis of the items d_i with $i \leq t$. Typical 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 elments 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 realtime constraints, such as a fixed limit on the time available for processing a data item, or a bound on the memory available for computation.

The framing 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).

Several algorithms to the task mentioned above with respect to these requirements have been proposed. Regarding the counting of elements and sets of items, a variety of different approximate count algorithms based on sketches of the data have been developed in [6, 5, 7]. For statistical model, estimators for quantiles have been presented in [11, 3].

Whereas a wide range of different methods have been provided for various streaming tasks, this work aims at providing an abstract framework to integrate these different approaches into a flexible environment to build a streaming analysis based upon the existing algorithms.

Existing Frameworks

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 [12]. 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* [13] as well as Twitter's *Storm* [10] framework do provide online processing and storage on large cluster infrastructures, but these do not include any online learning.

In contrast to these frameworks, the *streams* library focuses on defining a simple abstraction layer that allows for the definition of stream processes which can be mapped to different backend infrastructures (such as *S4* or *Storm*).

3 An Abstract Stream Processing Model

In this section we introduce the basic concepts and ideas that we model within the *streams* framework. This mainly comprises the data flow (pipes-and-filters [14]), the control flow (anytime services) and the basic data structures and elements used for data processing. The objective of the very simple abstraction layer is to provide a clean and easy-to-use API to implement against.

The structure of the *streams* framework builds upon thress basic parts:

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 modelled using a combination of different building blocks of the API (e.g. queues, services) or custom implementations of the facades.

3.1 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. We consider the case of continuous streaming data being modeled as a sequence of *data items*.

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.Data` interface.

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

Figure 1: A data item example with 3 attributes.

Figure 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 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.

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 2: 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 **??**.

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 **??**.

3.2 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 prespective of a process designer, the *stream* and *process* elements form the basic data flow elements whereas the processors are those that do the work.

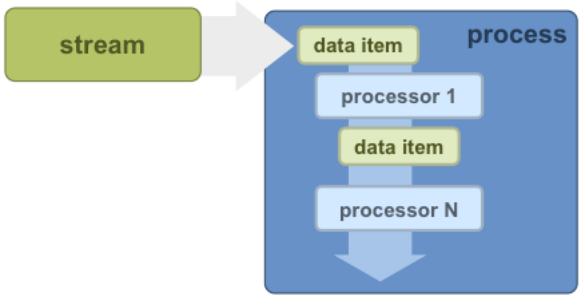

Figure 3: A process reading from a stream and applying processors.

The simple setup in Figure 3 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. We discuss the implementation of *active streams* and the resulting *push* oriented data flow in Section **??**.

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[1].

For communication between processes, the *streams* environment provides the notion of *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 4 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.

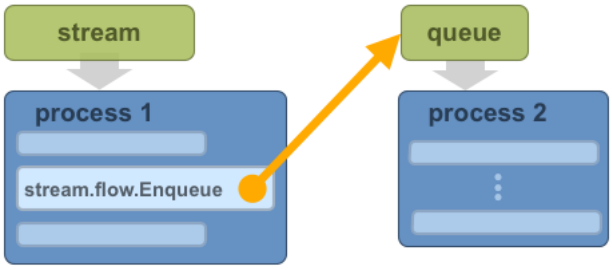

Figure 4: Two Processes P_1 and P_2 communicating via queues.

These five basic elements (*stream*, *data item*, *processor*, *process* and *queue*) already allow for modelling 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).

3.3 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 orthonogal 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 asynchronuous to the data flow. Figure 5 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

[1]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.

merged into the stream or services that can be queried for current statistical information (mean, average, counts).

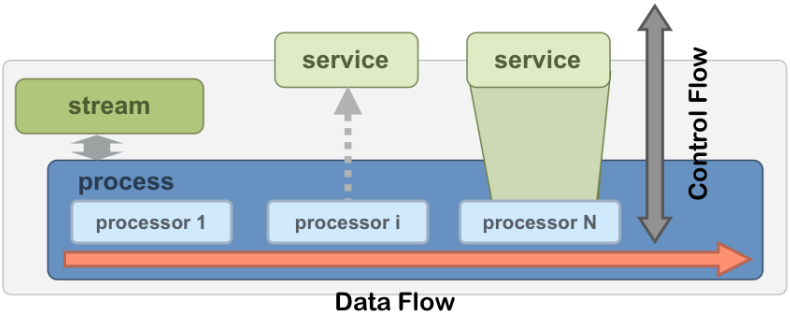

Figure 5: Orthogonal *data* and *control flow*. Processors may use services as well as export functionality by providing services.

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 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.

4 Designing Stream Processes

The former section introduced the main conceptual elements of the *streams* library. These basic elements are used to define stream processes that can be deployed and executed with the *streams* runtime environment.

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 3.

4.1 Layout of a Process Environment

The *streams* library follows the concept of runtime containers, known from Java's servlet specification and similar architectures. A single container may contain multiple processes, streams and services, which are all executed in parallel. A contains is simply providing the abstract environment and a joint namespace for these elements to execute.

An example for a container definition is provided in Figure 6. This example defines a process environment with namespace `example` which contains a single data stream with identifier `D` and a process that will be processing that stream.

```
<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 6: A simple container, defining a stream that is created from a CSV file.

The core XML elements used in the simple example of Figure 6 are `stream` and `process`, which correspond to the same conceptual elements that have previously been defined in Section 3.

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

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 implemenations can be found in Appendix **??**. The default stream implementations also allow for the use of a `limit` parameter for stopping the stream after a given amount of data items.

4.1.2 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 3.2.

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). The general behavior of a process is shown in the pseudo-code of Algorithm 1.

Algorithm 1 Pseudo-code for the behavior of a simple `process` element.

Require: A data stream S and a sequence $P = \langle f_1, \ldots, f_k \rangle$ of processors

```
function ProcessStream(S)
    while true do
        d := readNext(S)
        for all f ∈ P do
            d' := f(d)
            if d' = null then return null
            else
                d := d'
            end if
        end for
    end while
end function
```

4.1.3 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. They are read from the stream by the process and handled as shown in Algorithm 1.

The smallest unit of work that the *streams* library defines is a simple *processor*. A *processor* is essentially a function that acts upon a data item. The processors already available in the library are provided by Java classes, which implement the simple `stream.Processor` interface. The interface defines a single function as shown in Figure 7.

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

Figure 7: The `Processor` interface that needs to be implemeted to create new processor elements.

5 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.

5.1 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. This camera is recording electric energy-impulses which in turn is measured by sampling each pixel at a rate of 2 GHz.

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 physics stuff.

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).

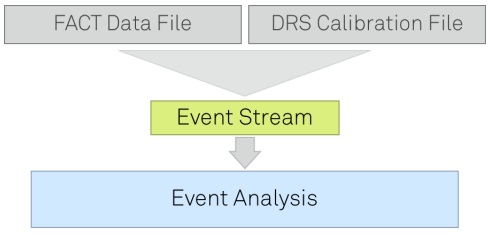

Figure 8: Stream-lined processing of events.

5.1.1 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 9 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 9: 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 1 lists all the attributes of an event that are currently provided by the `FACTEventStream` class.

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 1: The elements available for each event.

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.

5.1.2 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[2] (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.

Installing the FACT-Plugin

The FACT-Plugin is an extension for RapidMiner. The RapidMiner open-source software is written in Java and is available for multiple environments (Windows, Unix). It can be downloaded from `http://rapid-i.com`.

The FACT-Plugin is a simple Java archive (*jar*-file) that can be found at

`http://sfb876.tu-dortmund.de/FACT/`

To install the plugin simply copy the latest `FACT-Plugin.jar` to your RapidMiner `lib/plugins` directory. After restarting RapidMiner, the plugin will be loaded and all of its additional operators will show up in the operators list.

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

A 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.

A.1 Data Streams

A.2 Data Processors

A.2.1 Processors in `stream.flow`

A.2.2 Processors in `stream.data`

A.2.3 Processors in `stream.parser`

B 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.

B.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 B.1.2. Instructions for RedHat based systems such as RedHat, CentOS or Scientific Linux are provided in B.1.3.

B.1.1 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: jwall.org Software Repository <software@jwall.org>
Fingerprint: 175C 915F 51CA 8AA2 387B E3E8 48E6 B98D 1344 3F4A
```

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).

B.1.2 Installing on Debian/Ubuntu

There exists a Debian/Ubuntu repository at `jwall.org`[3] 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
```

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

The repositories and all packages within the repository are cryptographically signed with a GPG key. Please see Section B.1.1 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):

```
# 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
```

B.1.3 Installing 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 B.1.1 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.

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] 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.

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

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

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

[9] 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.

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

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

[12] 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.

[13] 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.

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