Technical Report

# A Survey of the Stream Processing Landscape

Version: 1.0

May 16, 2014

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**

The continuous processing of streaming data has become an important aspect in many applications. Over the last years a variety of different streaming platforms has been developed and a number of open source frameworks is available for the implementation of streaming applications. In this report, we will survey the landscape of existing streaming platforms. Starting with an overview of the evolving developments in the recent past, we will discuss the requirements of modern streaming architectures and present the ways these are approached by the different frameworks.

# 1 Introduction

Over the past years *Big Data* has become the predominant term of our information system era. Gaining knowledge from massive amounts of data is regarded one of the key challenges of our times. Starting with the problem to process the immense *volume* of data, *Big Data* has emerged additional properties: the *variety* of different types of data and the *velocity* in which new data is being produced. This is often referred to as the 3 *V*'s of the Big Data challenge [36]:

- *Volume*: the ability to process data in the range of terabytes and petabytes
- *Variety*: the need to combine data from all kinds of different sources and formats
- *Velocity*: the ability to keep up with the immense speed of newly generated data.

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 [24]. 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* [3] implementation of a map-and-reduce platform nowadays builds the foundation for various large-scale systems and has become the de-facto standard for Big Data processing with open-source systems. In [43] Sakr, Liu and Fayoumi survey the family of MapReduce systems along with their improvements. Emerged from the *Hadoop* platform has the *Zookeeper* cluster management sub-project [4]. Zookeeper is a fault-tolerant, distributed coordination service that has become one of the key core-components of modern distributed scale-out platforms.

### From Batches to Continuous Streams

Whereas the volume and variety have been the first encounters of the Big Data era, the need to address the velocity of data processing has become more and more important: As data is generated at higher speed, the validity of data is a quickly decreasing quality. For a very simple example, one may look at text data – long-term static web pages have been supplanted by more up-to-date weblogs. With blogging systems people started providing much more frequent updates, which have then been superseeded by micro-blogging in the form of twitter messages or status updates in social media. Where static pages had a validity of months or years, blogging pushed that periods down to days or weeks. The validity of twitter messages is often much less than days.

As a result, the processing of data needs to keep up with that evolvement of data and any results computed in today's systems must reflect that. Following the blog example, in mid 2010 Google changed its indexing system from pure batch-wise indexing to online updates of the search index in order to provide search results that reflect articles found within the last 10 or 15 minutes[1].

## 1.1 Emerging Streaming Platforms

Research in stream processing has come a long way from low-level signal processing networks to general purpose systems for managing data streams. Popular academic approaches for these data stream management systems (DSMS) are Borealis [7, 13], TelegraphCQ [19] and STREAM [28].

As shown in Figure 1, the field of stream processing approaches can be divided into *query-based* systems that emerged from database research; the *online algorithm research*, which has brought up sketch-based algorithms for computing approximate results in a single-pass over the data; and finally the *general purpose streaming platforms*, which provide means for implementing and executing custom streaming applications. These areas are not disjoint and benefit from each other.

### Query-based Systems

*Query-based systems* utilize a high-level query language to induce state automata from a user specified query, that are capable of online processing of streaming items. Depending on the query, the automaton will emit a stream of updated results for that query. Query languages are often tightly bound to SQL like dialects that extend a common language base with additional keywords for specifying window sizes for aggregates or intervals for emitting results.

[1] `http://googlewebmastercentral.blogspot.de/2010/06/our-new-search-index-caffeine.html`

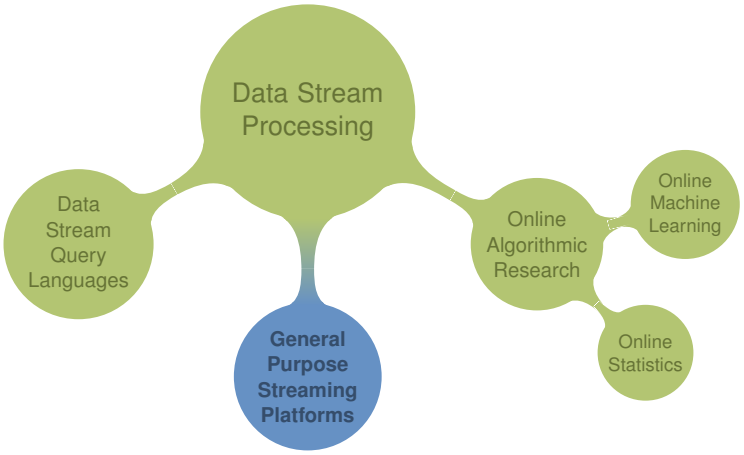



Figure 1: A partitioning of *data stream processing* into different areas, which have naturally evolved from previous database research as well as resource constrianed algorithmic engineering. The emerging *general purpose streaming platforms* are the subject of this article.

### Online Algorithmic Research

The field of *online algorithmic research* more generally explores different algorithmic aspects of computing results from unbounded, streaming data sources. A lot of the problems that are easy to solve on static data become intractable to compute on data streams, especially with the additional constraints of resource limitations like processing power or main memory. This area has brought up fundamental algorithms for simple problems such as counting elements in a stream [23, 27] or maintaining statistics over streams [29, 33, 38]. In parallel, learning methods that are capable of incrementally training models for prediction [16, 11, 25, 11] or clustering tasks [14, 18, 21, 30, 8] have been proposed.

### General Purpose Streaming Platforms

The *general purpose streaming platforms* have emerged from real world needs to process data continuously and being able to define custom **streaming applications** for specific business use cases. Whereas query based systems and the online algorithmic engineering focus on solutions to specific problems, the general purpose frameworks provide platforms for executing streaming applications whilst providing low-level means for application programming, scalability and fault-tolerance.

The integration of specific libraries and approaches like *query based* systems into the implementation of custom streaming applications integrates the outcome of the different fields into an environment that is powered by a general purpose streaming platform. An example for such integrative solutions is the 2013 DEBS challenge: The challenge was dedicated to process moving sensor data as fast as possible while computing and maintaining statistics over various sliding windows. This involved low-level preprocessing as well as high-level count aggregations over windows. The former is best implemented

in some programming language, pre-aggregating and filtering items to a lower frequency stream. The outcome stream can then be fed into a high-level query engine such as Esper [39] for computing online windowed statistics. Such an approach has been proposed in [26], combining low-level processing with a high-level query based system using the streams framework as general purpose stream processing framework.

## 1.2 Evolvement of General Purpose Streaming Frameworks

The trend to continuous online processing of real-world data has fostered a number of open-source software frameworks for *general purpose data stream processing*. Most of these systems are *distributed stream processing systems* (DSPS) that allow for distributed computation among a large set of computing nodes. With Yahoo!'s *S4* [41] engine being among the oldest such framework, we plotted a history of versions for most of the major stream processing platforms in Figure 2. Each dot is a new release whereas the circles are announcements or scientific publications related to the framework. In addition to the streaming platforms, we included the version dates of the Apache Hadoop project as the state-of-the-art batch processing framework into the chart. From that, one can clearly derive from this figure the shift of the requirement for data stream processing starting in 2011. The recent announcements of new frameworks such as *Samza* [5] or *MillWheel* [10] show that there still is an intrinsic need to expand the existing frameworks for better suitability.

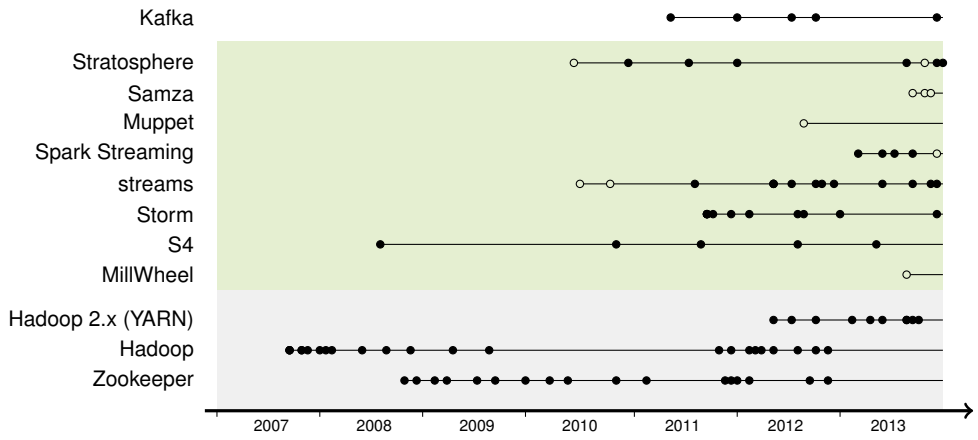



Figure 2: A history plot of versions of different stream processing platforms (green background). The lower part (gray background) denotes the Hadoop open-source Map&Reduce framwork for batch processing.

### Outline of this work

The obvious demand for general purpose streaming platforms motivates our survey of existing and emerging approaches towards real-time data stream processing. The plethora of different platforms and newly appearing software systems additionally gives rise to several questions:

- What differentiates general purpose streaming platforms from one another?
- Which properties does each platform provide? Which drawbacks?
- What is the best platform for a given use case?
- How does a *streaming application* look like for any of these platforms?

In this article we will review the requirements for stream processing that are tackled by these frameworks and give an overview of the general concepts which are inherited in all of them. Furthermore, we outline some example streaming applications and give a sample implementation within the context of each of the surveyed platforms.

According to that, the rest of this article is structured as follows: in Section 2 we start by providing an abstract concept of stream processing and review requirements to online data processing. Following that, we give a more detailed view of stream process execution engines in Section 3. Based on the aspects presented in this Section we investigate the properties of various stream framework implementations in Section 4. Finally we give a summary of the frameworks and outline the relation to our own streams framework implementation.

# 2 An Abstract View on Stream Processing

A natural perception on data stream processing is the modeling of data flows by means of a graph. Such a graph contains sources of data that continuously emit items which are processed by connected nodes that do some actual computation on the items.

Historically, this has been the core concept in message passing systems and follows a *data driven* programming concept. Essentially two types of elements need to be present: a data source element and an element that defines the processing of items emitted by the source as shown in Figure 3. In addition to that a common definition for the atomic data items that are emitted by sources and consumed by processing nodes needs to be defined.

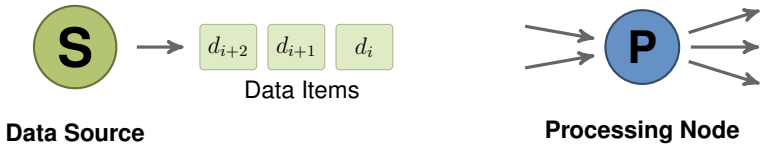



Figure 3: The concept of a simple *Data Source* and *Processing Node*.

With different terminologies, these elements are present in all the surveyed streaming platforms. As an example, within the *Storm* framework, sources are referred to as *Spouts* and processing nodes are called *Bolts*. The messages or data items passed between components in *Storm* are called *Tuples*.

### Streaming Applications as Data Flow Graphs

This simple notion of sources, items and processing nodes, allows for defining *streaming applications* by means of connected components within a graph. Such graphs are the application structure in all modern streaming platforms. For a very simple example, the graph shown in Figure 4 defines a streaming application that contains a single data source of log messages $m_i$, which is consumed by a processor node that extracts some tags $t_0, \ldots, t_j, \ldots, t_k$ from the incoming messages. The extracted tags are consumed by a collection of counter nodes, each of which maintains counters for the tags it processes. The counting nodes emit their aggregated counts to a final processing node which sums up the counts emitted by the counters.

The *Counter* nodes represent the executive elements of the streaming applications and pose the algorithmic challenges in the *online algorithm research* field. As an example, simple counting of elements in a streaming manner has been studied in [23, 27]. The objective of a general purpose streaming platform here is to provide an API to implement efficient algorithms for the task at hand and include it as processing node within a data flow graph definition, i.e. the design of a streaming application. The platforms task is then to execute instances of such a graph on one or more (in the distributed case) compute nodes and manage the distribution and routing of messages from one processing node to the other.

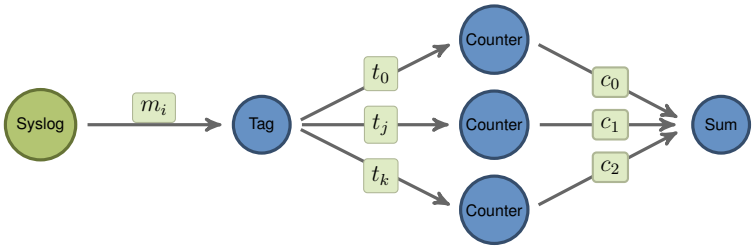



Figure 4: A simple graph for a streaming application that consumes data, and defines processing nodes for extracting new information and counting elements from that extracted new items.

## 2.1 Requirements of General Purpose Streaming Platforms

The execution of streaming applications is subject to various requirements that differ from traditional batch processing. In [44] Michael Stonebraker et.al. derived a set of general requirements for data stream processing engines that have become accepted distinctive features for streaming engines. The 8 proposed requirements listed in [44] are:

**R1** *Keep the data moving*
Process data without the need of storage to keep latency at an absolute minimum.

**R2** *Query using SQL on Streams*
Provide high-level means for building extensive operators.

**R3** *Handle stream imperfections*
Provide built-in features for handling missing values or out-of-order data.

**R4** *Generate predictable outcomes*
Guarantee repeatable outcomes and predictable results of executed processes.

**R5** *Integrate stored and streaming data*
The ability to combine streaming data with static external offline information.

**R6** *Guarantee Data safety and Availabilty*
Provide means for fault tolerance, resumption of execition and high availabilty.

**R7** *Partition and scale applications automatically*
Include means to distribute processing among multiple processors/CPUs/nodes.

**R8** *Process and respond instantaneously*
Achieve real-time response with minimal overhead for high-volume data streams.

Some of these requirements are inherently conflictive: providing guarantees for data safety and availability (**R2**) comes with a performance cost as it requires persistent states to be written to high available backend storage, which will introduce additional latency to data processing (**R1**, **R8**).

In addition to those computation oriented requirements, the notion of *usability* plays an important role for the acceptance and usefulness of a streaming engine by an end user. This requirement is only partly reflected in **R2** and we will additional include *usuability* as an additional quality to this survey.

Some of the requirements listed above are inherent to all surveyed stream processing engine: Todays streaming architectures are designed for moving data. In-memory processing is a central property and the field of online algorithmic research has for long time investigated the development of algorithms that run in sub-linear time and with fixed bounds for memory usage. By trading memory consumption for precision these approaches address the on-the-fly data processing without requiring expensive offline computations.

As we will outline in 4, the partitioning of data streams and scaling of the processing among multiple nodes is a key quality of the distributed streaming platforms and is being addressed by each framework in slightly different ways. The systems differ mostly in the level of transparency of how these features are provided to the user.

An interesting quality is the ability to deal with out-of-order data streams. Given the notion of a global temporal ordering of messages, the handling of global clocks in distributed systems has a long history of research (cf. [35]).

## 2.2 Usability and Process Modelling

Although the representation of streaming applications by data flow graphs is shared by all the surveyed streaming platforms, the frameworks differ in the way these applications are being created. A common denominator is the existence of a programming API that each of the frameworks provides. These APIs essentially provide an *environment* for custom user functions and further classes and functions to *programmatically create application graphs*.

Making use of these APIs, an application is often represented as some entry-level code that is submitted to the framework, upon execution creates a data flow graph and triggers the execution of that graph on the platform. Any modifications of the graph requires a recompilation and resubmission of the modified streaming program to the framework.

This programmatic creation of streaming applications allows for an extensive use of the frameworks features: apart from basic common functions, the framework mainly differ in the provisioning of features that an application developer may use.

### From Developers to Application Designers

The code-level approach for creating streaming applications introduces a burden for making direct use of such platforms by domain experts: with high expertises in their application domain, they are often confronted with a plethora of new concepts and features offered by the APIs of modern streaming architectures.

As the importance of stream processing raises among different application domains, the

question arises how domain experts can benefit from the emerging frameworks:

1. What features are required to be exposed to domain experts to make best use of streaming in their applications?
2. What is an appropriate level of abstraction for designing streaming applications by non-developers?
3. How may domain experts best incorporate their custom functions into a streaming application?
4. What level of abstraction does support the best re-use of existing code?

Based on these questions we extend the scope of the representation of streaming applications from low level programmatic creation to higher level application design. Figure 5 shows the stack of different design levels.

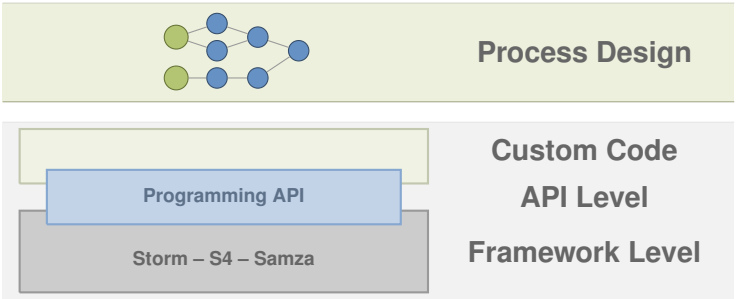



Figure 5: Different Levels for modelling streaming applications.

Apart from the very specific code level provided by APIs of the various streaming platforms, the *process design* level offers a much more high-level notion of creating applications. Concepts situated at this layer usually address the layout of the data flow graphs by use of pre-existing operators often accompanied with graphical tools. Examples for such approaches in batch processing are the RapidMiner tool suite [40] or KNIME [15] (both examples for application design in the data mining field).

## 2.3 Features of Modern Streaming Platforms

As mentioned above, the various streaming frameworks are geared towards the execution of streaming applications, each trading off features like the requirements listed above. Based on the aforementioned requirements we will survey the platforms with regard to the following categories:

- *Execution Semantics & High Availability*
- *Distribution & Scalability*
- *Usability & Process Modelling*

The execution semantics and high availability refer to the way messages are processed in the system. In the ideal situation, each message is processed *exactly once*. In case of node failures, the systems may re-send messages to compensate such failures by re-processing data items. Depending on the granularity of the API presented to the user, such failure handling is arranged in a transparent way.

The distribution and scalability aspect becomes important when facing large volumes of data. As a consequence of the arize of Big Data, many systems are inherently designed for the execution among multiple nodes which are managed in a centralized or de-centralized manner. The distribution of data streams among processes may have a direct impact on the organization of computations, as we will outline in Section 4.5.

Finally, the usability and process modelling aspect introduces an important facet when integrating a streaming platform in different real-world projects. Support for a transparent and easy to use environment that forsters a quick rapid-prototyping is often a requirement to allow for domain experts to make best use of a streaming framework.

# 3 General Purpose Streaming Platforms

Based on the abstract view of streaming applications in Section 2, we can identify basically two major functionalities that are provided by streaming platforms to allow for streaming applications to be run:

1. A *queueing* or *message passing* component, that provides communication between processing nodes
2. An *execution engine*, which provides a runtime or context for the execution of processing nodes.

In early versions, the two components have been tightly coupled, i.e. most execution engines use a fixed specific message passing system. For example, in the beginning of the *Apache S4* system, it completely relied on TCP connections for message passing. This has recently changed and some execution engines allow for using different queueing systems interchangebly.

Looking at the *distributed nature* of modern streaming platforms, a management system for distributing the execution engine and the message passing components onto a collection of connected cluster nodes is required as well. The Apache Zookeeper project has become the defacto standard of an open-source cluster management platform and serves as the basis for all the platforms surveyed in this article.

In the following we will first review some of the available open-source queueing and message passing systems in Section 3.1 and then provide a detailed description of the *stream execution engines* in Section 3.2. Based on this, we look into a number of implementations of popular streaming systems in Section 4.

## 3.1 Queueing and Message Passing

Each of the stream processing frameworks presented in Section 4 requires means of passing messages between the processing nodes of a streaming application. At the most low-level is probably the message transfer using TCP connections between nodes, where the engine will manage a directory of TCP endpoints of all available nodes and maintain the TCP connections between these. As an example, the S4 system in its early stages used direct TCP connections between its processing elements.

As applications scale to larger sizes, more sophisticated features are required. As an example using reliable multicast to distribute messages among multiple subscribers may be used to implement a *hot standby* fault tolerant mechanism (c.f. [32]). There exists a large number of different message passing systems, such as *RabbitMQ* [42], *ActiveMQ* [1], ZeroMQ (ØMQ) [9, 31] or the *Apache Kafka* message broker.

The two major messaging systems that are used within the stream processing frameworks surveyed in this article are *ZeroMQ* and *Apache Kafka*.

### 3.1.1 Direct Remote Method Invocation

The simplest form of sending messages across elements of a streaming application is a transparent *remote procedure call* (RPC) interface, which is inherently provided by a wide range of modern programming languages. As an example, the Java language includes the *remote method invocation* (RMI) system, which allows calling methods of remote objects. Such remote calls are usually mapped to simple client-server communications using TCP or other network protocols. For calling remote objects a broker is required, which provides a directory service listing the available remote objects.

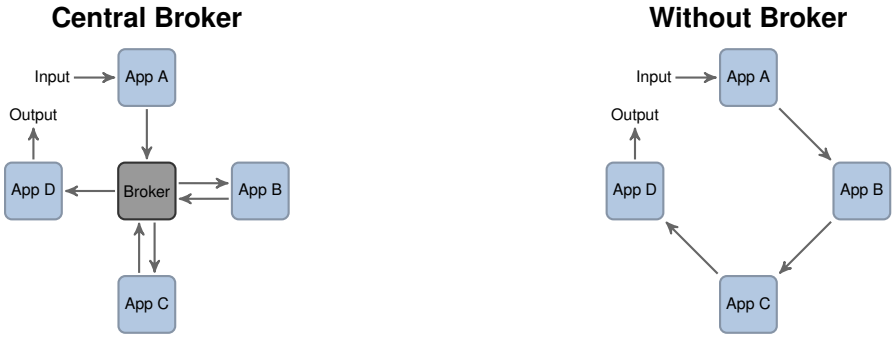
A popular programming paradigm based on remote procedure calls is the *Message Passing Interface* MPI. MPI was developed to be an abstract interface allowing to build massive parallel applications that execute among a set of nodes.

### 3.1.2 The `ZeroMQ` Queueing System

ZeroMQ (ØMQ) is a low-level messaging system that provides an API and bindings for various languages. It is an open-source library distributed under the Apache LGPL license. It abstracts the underlying transport protocol and provides reliable message passing, load balancing and intelligent message batching. The general aim of ØMQ is to build an API that is fast and stable to use, while allowing for a wide range of network topologies to be defined among the communicating components.

#### Scalability and Performance

Its lightweight design and minimum overhead results in high throughput performance and minimal latency. Despite its performance, ØMQ allows a wide variety of different message network models like the communication with a centralized broker (see Figure 6a) as well as direct communication of the participating nodes (Figure 6b).




(a) (b)

Figure 6: ØMQ messaging with a central broker (left) and direct communication without a broker (right).

In addition there are multiple ways of using a centralized broker as directory service for establishing the direct communication between nodes. All these network schemes are supported by the API of ØMQ.

### 3.1.3 Apache Kafka

Apache Kafka [34] has been designed as a reliable, distributed messaging system that follows the publish-subscriber pattern. It has recently been developed at LinkedIn as part of their streaming architecture and been donated to the Apache Software Foundation as an open-source messaging system. The Kafka systems is implemented in *Scala* and published under the Apache 2 License.

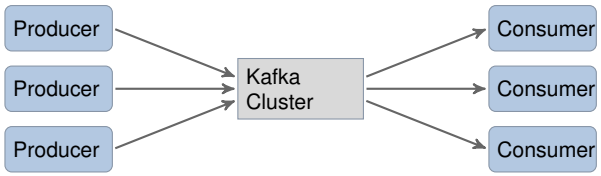



Figure 7: A distributed Kafka cluster providing message passing between producers and consumers.

Kafka provides a broker for managing queues, which are called *topics*. Producers of data streams may publish messages to topics and consumers may subscribe to topics to retrieve the messages from that topic. Figure 7 shows the central role of Kafka as a message broker.

A key design decision of Kafka over other messaging systems is, that Kafka explicitly stores all messages on disk. As streaming data is of serial nature, Kafka exploits the speed of *serial writing* and likewise *serial reading* from modern hard drives. Moreover, it benefits from page-caching of filesystem I/O in modern operating systems, without implementing its own caching stragegies. This makes Kafka a fast queueing system with a large persistent buffer.

The persistent nature of Kafka topics therefore directly allows to resume message processing of a topic at several points in the past: If the configured storage is able to hold a week of data on hard drives, processing of messages can be restarted from any time within that week. By this, Kafka directly supports easy means to the resuming of failed processing and offers a high degree of reproducibility.

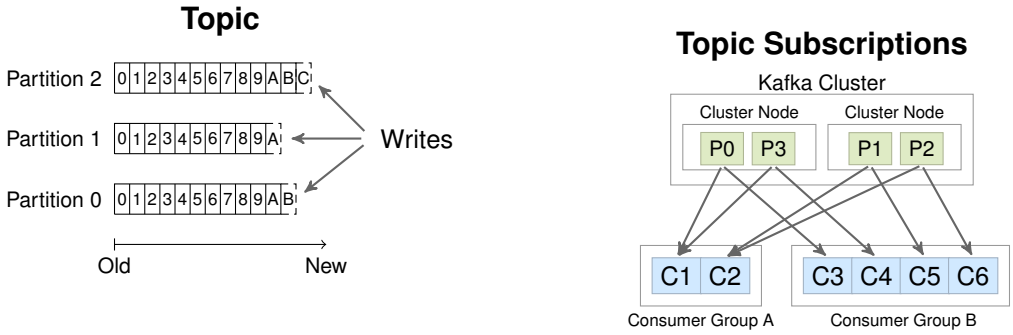
#### Kafka Clusters and High Availability

Kafka is designed to run in a cluster of machines, as a *Kafka cluster*, that is coordinated by an underlying Zookeeper system. Using Zookeeper, an election of a master node is performed and the other nodes become slave nodes.

Topics of a Kafka system can be created with a replication factor. The cluster will ensure that messages published to a topic will be replicated among multiple nodes within the cluster. Depending on the replication factor, this tolerates one or more nodes to fail without data loss. Consumers subscribing to a topic may use multiple brokers of the cluster to subscribe to a topic.

### Scalability and Partitioned Streams

As has been noted in Section 2.1, an important feature in modern streaming architecture is the ability to scale processing and message passing to a large number of nodes. Scaling out data processing relies on data partitioning and parallelization of tasks computing results on the data partitions. The Kafka system inherently divides the messages of a topic into disjoint partitions. Figure 8a shows the structure of a topic (or stream) in Kafka. A topic is split into a number of partitions to which new messages are appended. The partition to which a new message is appended is either determined by a specified *partition key* or by a random hash value. Using a partition key ensures that all messages related to a specific value are appended to the same partition. The ordering of messages is determined by the ordering within each partition. As an example, using username of a stream of twitter messages as message key for partitioning, will ensure that all messages of a user always appear in the same partition.




(a) (b)

Figure 8: Kafka topics divided into partitions (left) and consumer groups subscribed to a topic (right).

Consumers subscribe to topics and will receive all messages that are published for their topics. To exploit the maximum performance using parallelization, a consumer typically is reflected by a consumer group, which includes multiple consumer instances each of which is connected to one or more partitions of the topic. As can be seen in Figure 8, consumer group A has two consumers each of which connects to two partitions. The consumer group B consists of four consumer instances, which exclusively connect to a single partition of the topic.

This $n$-to-$m$ mapping of partitions to consumer instances allows for a high level of parallelisation and allows for a high degree of scalability of the message processing.

## 3.2 Stream Execution Engines

The core component of a general purpose stream processing system is a (distributed) runtime environment that manages the execution and distribution of processing nodes of a data flow graph. The processing nodes are connected by a queueing or message passing system as outlined in the previous section. The task of the execution engine is to instantiate the processing nodes and to execute the nodes within a runtime environment. The environment itself can be executing on a single machine or consist of a cluster of multiple machines. Usually, the environment provides a *worker context* or *executor* for each element of the data flow graph and these executors continuously run the code of those elements.

Figure 9 shows an abstract data flow graph on the left hand side and an instance of the graph with processing nodes being distributed and executed on two cluster nodes. As can be seen, the data source S has been instantiated on the upper cluster node and is being run within some executor. The executor provides a runtime context to the processing node instance. Likewise instances of processes $P_1$ and $P_3$ are being executed. For $P_2$ there exist *two* instances and output of $P_1$ is distributed among these executing instances.

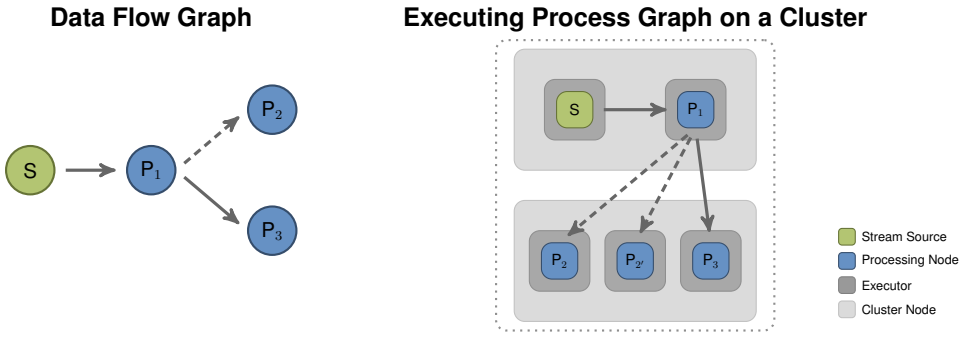



Figure 9: A distributed streaming engine executing a data flow graph on a cluster of nodes. Key to scalability is the partitioning of data streams and spawning of multiple copies of processes. In this figure process $P_2$ is spawned twice.

The cluster nodes of a distributed streaming runtime take care of supervising and spawning the required number of executors as well as balancing the executors among the available nodes (*load balancing*). In addition, the cluster nodes handle failing executors (processing nodes) and – depending on the fault tolerance model supported by the engine – may restart new instances of processing nodes and replay buffered messages.

The coordination of instantiating the processing nodes of a data flow graph and distributing these node instances among the executors on the different nodes of the cluster is usually being performed by a central master node.

### 3.2.1 Distributed Streaming Applications

The example in Figure 4 already demonstrates an important aspect in todays stream processing platforms: the ability to scale the computation by partitioning the data stream into substreams and handling these substreams with multiple copies of some processing

nodes. In the given example, the stream of tags is partitioned (e.g. by hashing the tag string) and dispatched among a set of *Counter* nodes. This approach translates the divide-and-conquer principle inherited in the Map-Reduce framework to the streaming setting.

Starting with the simple example, the scale-out affects the processing of the stream of *tags* that is produced by the *Tag Extractor* node, where a single instance of that node is contained in the graph. For additionally scaling the tag extraction part, the messages $m_i$ need to be partitioned by some discriminitive key $k(m_i)$ and dispatched among a set of tag extractor nodes.

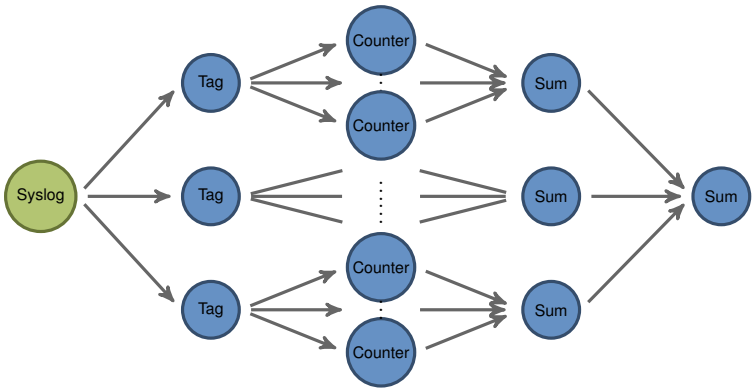



Figure 10: Data partitioning at the tag extraction stage, providing scale-out at an earlier stage for handling large amounts of messages $m_i$, which are partitioned by some explicit key $k(m_i)$.

This in turn opens various options for scale out as the outgoing tags can be routed to a large number of counters. An additional layer for aggregating the local sums is required to compute a continuous global sum over all tags of the partitioned stream.

The application graph shown in Figure 10 is a high-level representation of the application using data partitioning. Based on the partitioning and replication of processing nodes, the execution of the nodes can be distributed among several computing nodes of a cluster. For this, message passing between cluster nodes needs to be provided by the stream processing framework. For efficient deployment of the overall streaming application graph, the streaming platforms provide scheduling algorithms which take over the distribution of processing node instances among the cluster nodes.

### 3.2.2 Fault Tolerance in Distributed Streaming Applications

Large scale systems pose additional challenges to the underlying architecture. Among the most challenging problems is the fault tolerance of computation. Scaling systems to hundreds or thousands of nodes/machines makes hardware failures a day-to-day problem. As such, large scale distributed systems need to provide mechanisms to allow for the streaming applications to deal with system failures and recover computations.

Fault tolerance is usually implemented by replication and restart: Within the Map-Reduce system all data is replicated in blocks on different nodes and *map* tasks on these

blocks are restarted if the task is not finished after a specified time. In this case, the data is however *static* and permanently resides on hard disks.

In the stream setting this is slightly different: often, data cannot be stored permanently as a whole due to the high velocity and volume. Therefore only small parts of the data are stored for a short period of time and any attempts to ensure consistent operation over system faults is limited by this constraint. In [32] Hwang et al have broken down approaches to achieve high-availability into tree types of recovery guarantees:

- *Precise Recovery*
- *Rollback Recovery*
- *Gap Recovery.*

As strongest of all these guarantees, the *precise recovery* handles any effects of failure without information or precision loss. That is, the result of processing with failures occurring is identical to an execution without errors.

To the other extreme, the *gap recovery* matches the need to operate on the most recent data available. Failure may lead to (temporal) loss of data being processed, but processing will continue as soon as new data arrives. This situation is found in temporal outages of sensors in a sensor network and any shortcomings may need to be taken care of by the application - e.g. by interpolating missing measurements.

The *rollback recovery* is probably the best known approach that is inherent in the ACID paradigm of transaction oriented commits and rollbacks in traditional database systems. The following Figure 11 shows two examples for fault tolerance handling using commits with replication of state and restart (*rollback*). The simple *Tag Extractor* node on the

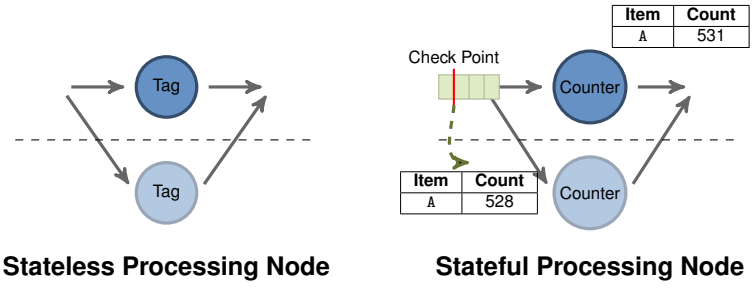



Figure 11: Fault tolerance handling by replication: Stateless processing nodes are much cheaper to replicate. Stateful nodes require state persistence, check-pointing and spooling of items.

left does not require a state as it does preprocessing on a single item only. Replication can easily be done by creating a new instance of the same node on a different machine and routing data items to that replica. In the case of a *stateful* processing node, such as the *Counter* node on the right hand side, the state needs to be check-pointed, made persistent and a replication of the node needs to resume the stream processing at the last

check-point, requiring a replay of the data that has been processed by the failing node since that very last check-point.

The frameworks we survey in this article differ with respect to the transparency how they offer fault tolerance and high-availability guarantees to the user: Most of the frameworks do provide failure detection (e.g. by timeouts) and replay of data. By signaling that to the user code, it is possible for custom code to handle deduplication of replayed data and recover from the last commit point. Some of the frameworks directly provide deduplication of messages and even offer state management functions through their API. This allows for freeing custom code from state handling in the first place, enabling the framework to fully provide a transparent failure recovery.

The buffering of replay data differs among the frameworks as well: Whereas Storm targets at *rollback recovery* by using acknowledgements to control in-memory buffering for data replay between check-points, the *Samza* system uses *Kafka* as message broker, which comes with optimized *serial writes* on disks to maintain a copy of the stream on different nodes.

Opposed to that, the *S4* framework in its early stages did operate at the *gap recovery* level and tolerates failures by requiring re-starts to be resumable with acceptance of loss of intermediate results or state. We will discuss the details for each framework in Section 3 in detail.

### 3.2.3 Programming API

To allow for the implementation of custom processing nodes, the stream execution engines provide programming APIs that wrap the context of the executors. Based on the features provided by the engines, these context provide methods to signal persistency of state or submit messages to other queues/processing nodes.

The programming API therefore embeds the custom code into an execution context, that provides the distributed execution and communication and means for checkpointing to ensure fault-tolerance can properly be provided by the execution system. The programming APIs of the streaming platforms differ in their functional power: whereas Storm does not provide any utility functions for managing state, MillWheel or streams provide interfaces to handle the state of computations completely outside the scope of the user code.

# 4 Stream Processing Frameworks

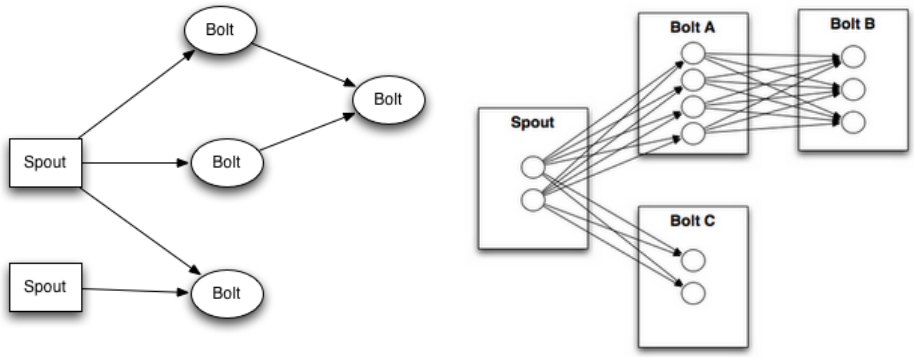
In the previous Section 3 we gave a general overview of the structure of stream processing platforms. As already noted in the introduction, a large amount of different implementations exist each of which focuses on different aspects in stream processing.

In this section we survey a set of popular streaming platforms.

## 4.1 Apache Storm

The Storm project is a distributed stream processing engine that has initially been started by Nathan Marz and further been developed at Twitter. It is written in *Java* and *clojure* and currently being incubated into the Apache Software Foundation.

Storm provides a notion of a *topology* that describes a streaming application. Such a topology conists of *spouts*, which emit data and *bolts*, which consume data from spouts, do some processing and may produce new data as output. Figure 12a shows a simple topology with connected spouts and bolts that represent a streaming application. A topology within Storm is defined by a Java program that creates a topology object and submits it to a storm cluster.




(a) (b)

Figure 12: A simple Storm topology (left)and groupings (right).

A storm cluster is a set of at least one *nimbus* node and one or more *supervisor* nodes. The *supervisors* provide an environment to execute spouts and bolts whereas the *nimbus* is a central coordinator that balances the instances of the spouts and bolts among the supervisors. Storm uses the Apache Zookeeper system to provide the coordination of its cluster nodes.

### Messages and Message Passing

The messages passed within Storm are called *tuples*. A tuple is a set of values for a predefined set of fields. Each spout and bolt defines the fields of the tuples it emits statically in advance. All tuples need to be serialized into a binary form before transmission to

other components. This serialization is handled by the *kryo* library, which provides a fast serialization of Java objects. To pass messages between the elements of a topology, Storm uses the ØMQ messaging system.

Storm is designed to handle all processing in memory without disk storage. The low latency of message passing using the high-performance ØMQ system directly yields towards requirement (R1).

### Distribution and Scaling

In addition to defining *spouts* and *bolts* and connecting them within a topology, for each bolt the number of *worker threads* may be specified. This results in one or more copies of a bolt to be instantiated and executed. By using *groupings* of the data streams (i.e. the connection between elements), this allows for splitting up a stream of tuples by custom groups and delegating these to different copies of a bolt. Figure 12b shows the distribution of data streams among instances of a bolt. This allows for scaling up computation by distributing a high-volume stream among multiple instances of a bolt, which may in turn be distributed among multiple nodes of a storm cluster.

The data grouping itself needs to be manually defined within the topology and remains static. The cluster then automatically manages the distribution of the bolt instances amond all available cluster nodes and routes the data elements accordingly. The distribution of streams among multiple instances of bolts offers a high degree of scalability (R7).

### Execution Semantics & High Availability

Storm features several execution semantics. In its default mode, all tuples are processed in an *at-most-once* manner, i.e. tuples are sent to bolts and will not be re-send if a bolt fails.

In addition, Storm optionally provides an *at-least-once* processing of tuples, i.e. it ensures that an item is processed by a bolt at least one time. This is achived by buffering tuples sent to a bolt in main memory and releasing these tuples as soon as the receiving bolt has acknowledged their correct processing. In case such an acknowledgement is not received within some time limit, the tuples are sent to the bolt again. This may result in tuples being processed multiple times as well as tuples arriving out-of-order. The *at-least-once* semantic requires the code of the bolt to explicit send acknowledgements.

As the strongest processing guarantee, Storm supports the *exactly-once* processing of tuples. With the acknowledgements of processed tuples and additional state persistency of bolts, this allows for a transaction oriented processing. For that, the code of the bolt is required to maintain its state in some external storage and allow for reloading its state at instantiation time. Thus, if a bolt fails, Storm is able to create a new instance of that bolt, which will restore its state from some external storage and Storm will handle the replay of the tuples that have not yet been acknowledged by the failed instance of the bolt. The storing of the bolts state as well as restoring the state upon restart is required

to be coded by the developer. Storm does not automatically save and restore states of a bolt.

With this behavior, Storm supports the implementation of the *rollback recovery* mechanism described in [32], requiring a strategy of commits/rollbacks to be provided by the programmer of the bolt. The *exactly once* processing of tuples ensures the predictable outcome and reproducability of the execution of topologies (R4).

Storms message processing guarantees may even be bound to transitive dependencies between messages. That is, if some tuple $m$ is processed and the processing at some node results in new tuples $m'_1, \ldots, m'_k$ being emitted, then $m$ is regarded as *fully processed* of all the related tuples $m'_i$ have been processed. The successful processing of a tuple is noted by an active acknowledgement, sent from the processing node to the node the tuple originated from. This may result in dependency trees among the topology as shown in Figure 13. Until a tuple is acknowledged, the sender of the tuple will buffer it in main memory for a potential resubmission. If processing of a tuple $m'_i$ fails, Storm will re-send the tuple as part of its recovery mechanism. By backpropagation of the acknowledgements by the processing nodes, the tree can finally be fully acknowledged back to the root tuple and all the tuples of the tree can be discarded from the recovery buffers as *fully processed*. The use of acknowledgements within the topology obviously adds processing overhead to the overall system. Therefore it is left optional to the user to make use of this feature or tolerate a possible lossy or incomplete processing of messages.

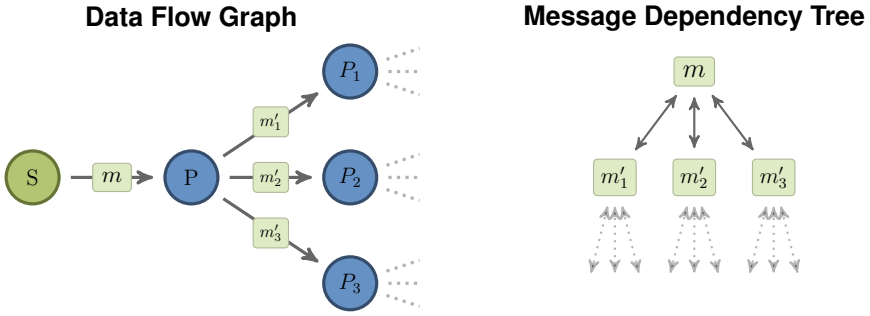



Figure 13: Storm topology and *full processing* of dependent messages.

High availability of Storm applications is achieved by running multiple supervisors and managing them on top of a fault tolerant coordinator (Zookeeper). The Storm systems detects failures by missing acknowledgements (i.e. through timeouts) and connection errors and employs restarts if instances of bolts on different supervisors. Apart from the aforementioned message acknowledgements, Storm does not provide any features to achieve persistency of the state of processing nodes. To ensure that the processing of messages resumes properly in case of a fault, the nodes are required to be implemented such that state is made persistent in some (fault tolerant) storage and acknowledgements are sent as soon as the state has been reliably stored.

New supervisors may join a cluster at any time and a rebalancing of the topology element instances allows for a *hot moving* of components to other machines. By ensuring that the supervisors themselves are running under process supervision, this creates a fault tolerant stream processing platform.

### Usability and Process Modelling

The core structure of a Storm application is the topology of spouts and bolts. This topology defines the data flow graph of the tuples and additionally allows for the user to define groupings of the tuples to split high volume data streams into substreams that are processed by multiple instances of a bolt.

The topology itself is defined in Java or clojure code and the user provides a Java program that creates the topology and submits it to the cluster. Regarding the usabilty levels defined in Section 2.2, Storm applications are created on the *Custom Code* level by using the API provided with Storm.

## 4.2 Apache Samza

The *Samza* framework is a stream processing execution engine that is tightly built around the *Kafka* messsage broker. Samza has originally been developed at LinkedIn and has recently been donated to the Apache Software Foundation. It is implemented in the *Java* and the *Scala* programming languages.

The processing nodes in Samza are represented by *Samza Jobs*. A job in Samza is connected to a set of input streams and a set of output streams. Thus, a job contains a list of input descriptions, output descriptions and a *Stream Task* that is to be executed for each of the messages from the input. When a job is being executed a number of Stream Tasks of the job are instantiated and provided to *Task Runners*. These task runners represent the execution context of the task instances and are managed by the Samza runtime system.

The philosophy of the Samza framework defines jobs as completely decoupled executing tasks that are only connected to input and output streams. Any more complex data flow graphs are created by submitting additional jobs to the Samza cluster which are then connected by the streams (topics) provided by the messaging systems (e.g. Kafka).

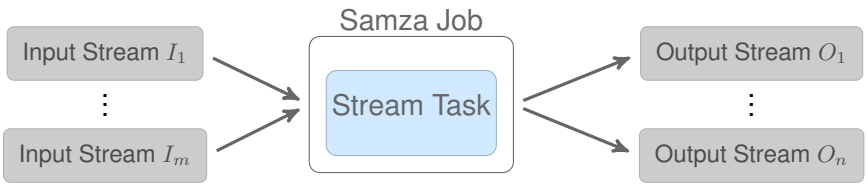



Figure 14: A Samza Job with $m$ input streams and $n$ output streams. The job is executing a *Stream Task* that is provided by the user/developer.

### Execution Semantics & High Availabilty

As Samza uses Kafka as message broker[2], all messages are stored on disk, providing persistence of the streams consumed and produced by Samza's stream tasks. This allows for a restart of failed tasks by resuming at the last valid position in the data stream that is provided by Kafka. Building on top of Kafka, Samza does provide an *at-least-once* semantic for the processing of messages. Any further message guarantees (i.e. *exactly-once*) requires custom handling, e.g. by keeping track of duplicates and discarding messages that have already been processed.

Instead of implementing its own, fault tolerant process execution engine (i.e. like Storm), Samza provides a context for its jobs by means of *Task Runners* and uses the Hadoop YARN platform to distribute and execute these Task Runners on a cluster of machines. Hadoop YARN is a continuation of the Apache Hadoop framework and provides a high-level cluster API of loosely coupled machines. Worker machines in such a YARN cluster run a *Node Manager* process which registers to a central Resource Manager to provide computing resources. A YARN application is then a set of executing elements that are distributed among the *Node Manager* processes of the cluster machines. Hadoop YARN provides abstract means for handling fault tolerance by restarting processes.

For executing a Samza job, the job elements are provided to a Samza Application Master (AM), which is allocated by requesting the Resource Manager to start a new instance of the AM. The AM then queries the registered Resource Managers to create YARN containers for executing Samza Task Runners. These Task Runners are then used to run the Stream Tasks of the Samza job. As the allocation of distributed YARN containers is provided by the Resource Manager, this results in a managed distributed execution of Samza jobs completely taken care of by Hadoop YARN.

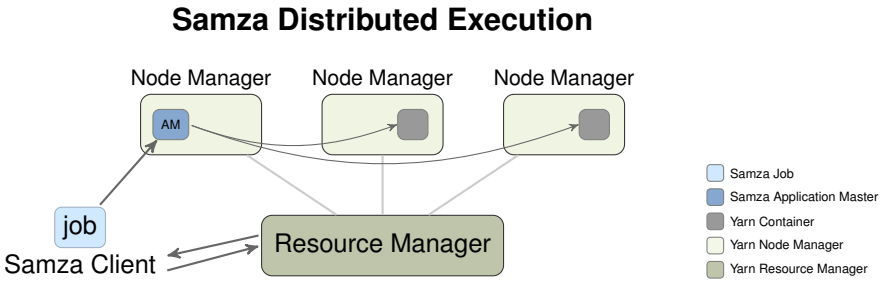



Figure 15: Architecture of the Samza job execution on Hadoop YARN. The Samza client requests the instantiation of an Application Master, which then distributes copies of the task of a Samza job among YARN containers.

### Scalability and Distribution

A Samza job that is defined by a Stream Task $T$ and connected to an input stream $I$ will result in the parallel execution of multiple instances of the job task $T$ for distinct parts

[2]Use of Apache Kafka as message broker is the default setting. Samza claims to support different messaging systems as replacement.

of the stream $I$. The partitioning of data streams (i.e. Kafka topic) in Samza is provided by the partitions of topics of the Kafka messaging framework (see Section 3.1.3). With the splitting of data streams into sub streams Samza provides a level of parallelization by spawning multiple copies of the Stream Task contained in the Job and executing these copies in several Task Runners. Each of the Stream Task instances is connected to one or more partitions of the input and output streams. Figure 16 shows the definition of a Samza Job connected to a single input stream. When executing, the Samza system will fork copies of task $T$ for processing the partitions.

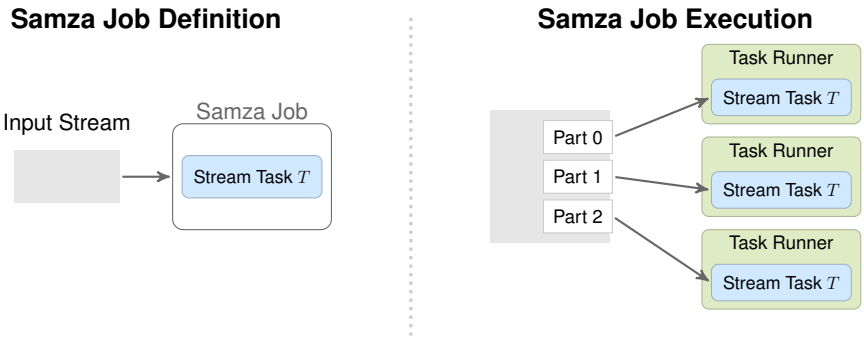



Figure 16: Partitions of a stream being connected to multiple instances of a Stream Task executing in several Task Runners.

The distribution of the task execution and the messaging is handled by the Hadoop YARN system. Samza uses Task Runners in YARN containers of a distributed YARN cluster to execute the Stream Task instances. On the other hand, the distributed Kafka message broker system provides the replication and availability of Kafka topics among multiple nodes and each task can be directly subscribed to a near partition of the stream it is connected to.

### Usability and Process Modelling

The modelling of data flow graphs within Samza requires the implementation and deployment of Samza jobs by custom Stream Tasks. For this, Samza provides a Java API for the implementation of *producers* and *consumers*. The code for a job is then compiled and bundled in a Java archive file, which is submitted to the execution environment (YARN) by the Samza client application.

With the Samza philosophy of decoupled jobs, there is no notion of a complete data flow graph being modeled as a single entity. Instead, users are responsible for setting up and deploying each job on their own.

## 4.3 S4 – Distributed Stream Computing Platform

The S4 platform is a distributed stream processing framework that was initially developed at *Yahoo!* and has been submitted to the Apache Software Foundation for incubation into the Apache Software Repository. It is open sources under under the Apache 2 license.

*Note:* According to the Apache incubator report of March 2014, S4 has considered to be retiring from incubation as the community is inactive and development efforts have deceased.

The S4 system uses a cluster of *processing nodes*, each of which may execute the processing elements (PE) of a data flow graph. The nodes are coordinated using Apache Zookeeper. Upon deployment of a streaming application, the processing elements of the application's graph are instantiated at various nodes and the S4 system routes the events that are to be processed to these instances. Figure 17 outlines the structure of a processing node in the S4 system.

Each event in the S4 system is identified by a key. Based upon this key, streams can be partitioned, allowing to scale the processing by parallelizing the data processing of a single partitioned stream among multiple instances of processing elements. For this, two types of processing elements exist: *keyless PEs* and *keyed PEs*. The keyless PEs can be executed on every processing node and events are randomly distributed among these. The keyed processing elements define a processing context by the key, which ensures all events for a specific key to be routed to that exact PE instance.

The messaging between processing nodes of an S4 cluster is handled using TCP connections.

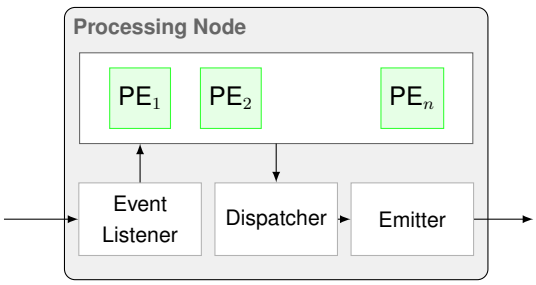



Figure 17: The structure of an S4 processing node executing multiple processing elements (PEs). A node may execute the PEs of various applications.

### Execution Semantics & High Availability

S4 focuses on a *lossy failover*, i.e. it uses a set of passive stand-by processing nodes which will spawn new processing elements if an active node fails. No event buffering or acknowledgement is provided, which results in *at-most-once* message semantics. As noted in [41], the newly spawned processing elements will be started with a fresh state and no automatic state management is provided.

Based on information on the latest 0.6.0 version[3], an automatic check-pointing mechanism has been implemented, which allows for processing elements to be periodically checkpointed by serializing the Java object to a backend storage. This checkpointing process is configurable to be triggered by time intervals or by message events.

#### Scalability and Distribution

With its concept of key-based partitioning of stream events, S4 follows the same principles as the other frameworks: scalability is gained by parallel processing of messages where the partitioning key defines the context of each of the parallel processing element instance.

By using the Apache Zookeeper system, S4 builds upon a decentralized cluster management of nodes. However, as of [41], the number of processing nodes within an S4 cluster is fixed, i.e. no additional nodes can be added dynamically.

#### Usability & Process Modelling

The S4 system uses a dependency injection based approach which is based in the Spring Framework [2]. Spring provides an XML based configuration that allows for users to define processing elements and their interconnection to be specified in an XML file.

### 4.4 MillWheel

MillWheel [10] is a distributed, fault-tolerant stream processing framework developed by Tyler Akidau, Alex Balikov et al at Google. It is a low-latency data processing framework for streaming applications that is widely used at Google, but at the time of writing, there does not exist an open-source implementation[4].

The design goals of MillWheel are:

- *minimum latency*, no intrinsic barries for data input into the system
- *persistent state abstractions* available to user code
- *out-of-order* processing of data with *low watermark* timestamps inherently provided by the system
- *scale out* to a large number of nodes without increasing latency
- *exactly-once* processing of messages.

Applications in the MillWheel system are defined as data flow graphs of data tranformations or processing nodes, which are in MillWheel terminology called *computations*.

[3]For information beyond the official publication [41], please refer to `http://incubator.apache.org/s4/doc/0.6.0/`.

[4]Google did not release an open-source implementation of its Map-Reduce framework either. Apache Hadoop is an open-source community implementation of the Google Map-Reduce framework.

The topology of computations can be changed dynamically, allowing for users to add or remove computations from the running system without the need of a restart.

From the messaging perspective, MillWheel follows the publish-subscriber paradigm: a stream is some named channel in the system to which computations can subscribe for messages and publish new result messages. As stated in [10], messages are delivered using plain remote procedure calls (RPC), which indicates that no queueing is included. Messages are associated with a *key* field, which is used for creating a computation context. Computations are performed within the context of that key, which allows for a separation of states and parallelization of processing among different key values. A computation that subscribes to a stream specifies a *key extractor* that determines the key value for the computation context.

In the example in Figure 18, the computation A will process messages which are aligned by the key `search query` (e.g. values like "Britney Spears",...), whereas computation B receives the same query objects grouped by the `cookie id` value of the records.

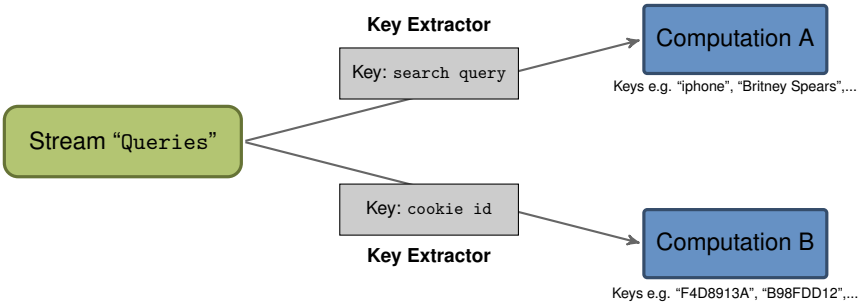



Figure 18: Two computations subscribing to a stream called `queries`. Each computation specifies a *key extractor* that defines which *key* value the messages contain.

### Execution Semantics & High Availability

The MillWheel system provides an API for writing custom computations, which offers abstract access to state managing and communication functions. By exclusively using these functions for state and communication, the user code is freed of any custom state handling and the system will keep track of the computations state (using high availability storage). This allows for the system to run computations in an idempotent manner. The computations are provided with a persistent storage that is unique per key, per computation.

Moving the state management entirely to the MillWheel API allows for providing a restart failure handling policy. By combining this with an automatic duplication handling by the system, MillWheel guarantees an *exactly-once* processing of data records.

A distinctive feature of MillWheel over the other frameworks is the focus on processing *out-of-order* messages. All messages in MillWhell are represented as triples

(key, value, timestamp)

where *key* is some arbitrary meta data field, that defines the semantic context of a message as described above. The *value* field is some byte string that can contain a custom payload i.e. the complete record for that message. Finally, a *timestamp* marks the time that tuple is associated with. Based on the timestamp values, the MillWheel system computes the low watermarks.

Let A be a computation and $\tau(A)$ be the timestamp of the oldest, not yet finished work of A. The low watermark $L_A$ of a computation A is recursively defined as:

$$L_A = \min \Big[ \, \tau(A) \, , \, \min\{L_C \mid C \text{ outputs to } A\} \, \Big].$$

The MillWheel system manages a global low watermark among the processes of the application which advances with the progress of completed work. This allows for out-of-order events to be transparently managed by the application – it only needs to rely on the clock provided by MillWheel.

### Scalability and Distribution

The MillWheel execution engine consists of a central (replicated) master node, that manages the blancing of computations among a set of slave nodes. As outlined in 3.2.1, the central aspect of scaling stream processing is the partitioning of data streams among multiple instances of a process/computation. For this, MillWheel exploits the message context by the extracted keys, where each computation is performed in the context of a particular key value. The key space for the computations is split into key intervals and the intervals are then assigned to the slave nodes. This determines how messages need to be routed by their key to the correct computation instance. The key intervals in MillWheel can be dynamically merged, split and moved among the slave nodes to adapt to increasing or decreasing work loads.

Persistent state is backed by highly scalable and distributed systems like Google's BigTable [20] or Spanner [22]. These systems are designed in a fault-tolerant manner themselves and provide reliable storage.

### Usability and Process Modelling

As of [10], no information is provided about the programming API or the way streaming applications in MillWheel are defined. The authors only provide a small excerpt of the definition of key extractors in a JSON like syntax. The sample implementation of computations in [10] suggests a C++ based implementation of user code.

MillWheel supports a dynamic change of the data flow graph, allowing users to add or remove computations without restarting the system. Similar to the Samza framework described in Section 4.2, components/computations can therefore dynamically subscribe/unsubscribe to channels.

## 4.5 Stratosphere

The *Stratosphere* [6] project is a DFG funded research project aiming at big data analytics with low latency. It extends the Map&Reduce paradigm by a declarative contract-based programming model (called PACT) that allows for compile-time optimization of program parallelization. *Stratosphere* focuses on streaming data and compiles PACT programs into data flow graphs of the *Nephele* execution engine [45].

Stratosphere is written in the Scala and Java programming languages and runs on a Java virtual machine. It is an active open-source project that at the time of writing is being incubated into the Apache Software repository.

Applications in Stratosphere are implemented as Java or Scala programs using the abstract API provided by the system. This allows users to create data flow graphs using provided or custom functions. Each of the functions of the graph, which are referred to as *user-functions*, specifies an *input contract* and may specify additional *output contracts*. These contracts are the core idea of the PACT programming model and define properties of the input and output required and produced by the user-functions. Thus, each possible function inherently provides hints about its input and output partitioning.

Based on this partitioning hints, a compile is applied to generate an optimized execution plan for the given PACT program. The resulting data flow includes the user-function nodes as well as channels and nnodes for partitioning of the data into parallelisation units, which reflect the scope of data required by the user-functions.

The resulting application graph is submitted to the local or distributed Stratosphere cluster which analyzes the graph and applies its

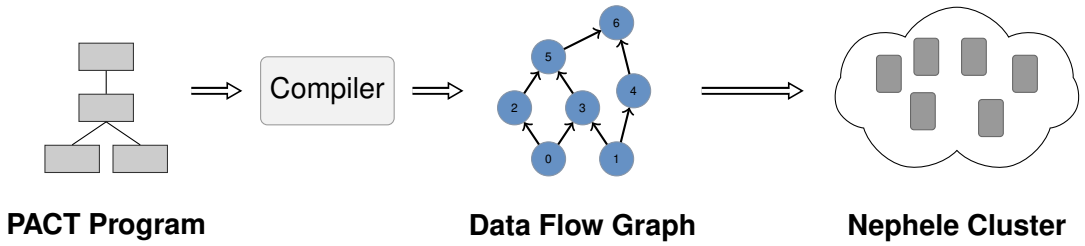



Figure 19: Compilation of a PACT program into a data flow graph. The data flow graph is statically optimized at compile-time, based on the input- and output-contracts of the user-functions.

### Execution Semantics & High Availability

The programming model used by Stratosphere focuses on an *exactly-once* messaging, that is completely hidden to the user. Users write their PACT programs typically in Java or Scala without the need to take care of deduplication or output errors. The fault-tolerant behavior is provided by the *Nephele* execution engine, that the compiler of Stratosphere is designed for. Though the authors in [6] note, that other execution engines might be used as well, Stratosphere is currently designed to work with Nephele.

### Scalability and Distribution

The ability to scale computation to a large number of nodes is one of the central aspects of the *Stratosphere* system. As in the other frameworks, scalability is gained by data partitioning and parallelization. For this partitioning Stratosphere explicitly introduces the notion of *parallelization units* (PU). Though this follows the same principles like the groupings in *Storm* or the partitionings of topics in *Samza/Kafka*, the idea of the partitioning units is more tightly bound to the user functions or processors of the data flow graph.

With the PACT programming principles, each user function defines its input and (optionally) output data with regard to the split into parallelization units. This explicitly defined extra information about the streaming functions can then be exploited by the compiler to generate data flow graphs from PACT programs that are optimized towards parallel processing of the PUs among multiple nodes of the Nephele compute cluster system.

The Nephele system that Stratosphere builds up on is a distributed system of computing nodes which are coordinated by a central master. Nephele supports fault-tolerance execution of processes among its nodes, featuring re-start and data deduplication. This provides an execution environment to Stratosphere that ensures reliable processing with exactly-once messaging semantics.

## 4.6 The streams Framework

The streams framework is an open-source streaming environment written in Java. It has been developed by Christian Bockermann at the Technische Universität Dortmund and addresses the design of data flow graphs by means of an XML abstraction language. The streams framework consists of a high-level API and an execution engine written in the *Java* language. It is an open-source framework published under the GNU AGPL license.

The main objective of the streams framework is to provide an abstraction that offers a simple API for implementing general streaming functions and allowing for the orchestration of streaming applications by the provided XML definition language. This aims at lowering the barrier of accessing streaming functionality to a wide range of users, enabling domain experts to use streams directly in their application by simply designing data flows without thinking about implementation issues.

The high-level API defines all elements required for building streaming applications such as *sources*, *processing nodes* and a data format for the representation of messages. As a distinctive feature compared to other stream processing frameworks, the streams API is not directly coupled to the streams runtime, but aims at providing an abstraction layer for different execution engines. Thus, it does aim at the design of generic data-flow graphs that can be mapped to other execution engines. As an example, applications defined in streams can be run on the *Storm* engine without changes of the application code [5]. Figure 20 shows the mapping of an XML definitions to the streams-runtime or to a Storm topology.

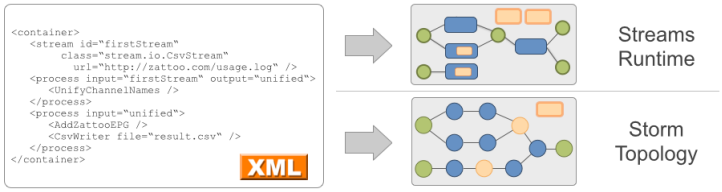



Figure 20: Modelling an application and compiling it into the streams runtime or a Storm topology.

### Process Modelling using Streaming Functions

The modelling approach of streams over the other execution engines differs in the sense, that within streams , another design-layer exists that encapsulates the notion of streaming functions by means of so-called *processors*. Thus, processing nodes of a streams application consists of a sequence of processors as shown in Figure 21. The processors are streaming functions which are combined in a pipeline to form a processing node in the data flow graph.

Processors are therefore components that can be used to model the behavior of processing

[5] The processing nodes of streams are then wrapped in Bolts of the Storm framework. This functionality is provided by s *streams-storm* wrapper library

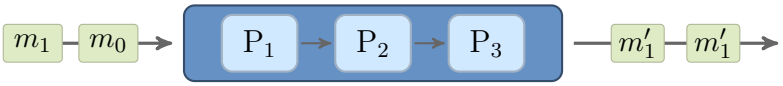



Figure 21: A processing node in the streams framework.

nodes. By providing a library of generic or domain specific processors, a user is allowed for combining processors (e.g. provided by a library) to create processing nodes that perform the required steps of computation. Following the dependency injection pattern, custom streaming functions can easily be provided as plain Java Bean classes and directly be included in the process definitions. In addition, streams supports the implementation of streaming functions as inline-code using JavaScript.

A crucial aspect in this finer grained representation of processing nodes is the **integration of external libraries** directly within the modelling of the data flow graph. As an example, the streams framework provides support for integrating online learning algorithms of the MOA [17] library as processors in the application design.

In addition, several application specific libraries have been implemented on top of the streams API, which provide processors for video analysis, system log data processing, FACT telescope data and much more.

### Execution Semantics and High Availability

The default execution engine of streams (*streams-runtime*) does not provide any fault tolerance features. Executing processing nodes apply all their streaming functions to each message they receive in an *exactly-once* manner. The streams-runtime is designed with the *let it crash* or *fail fast* philosophy in mind. This paradigm has been investigated by Armstrong in [12] to built highly robust software applications (with his Erlang language). The main reason for choosing this path as the default mechanism in streams is, that the streams runtime is intended as an embeddable runtime focusing on single node execution. Though it features remote queuing and communication with other instances of the runtime, its primary focus is on executing on a single machine.

When running streams applications using the streams runtime under supervision using auto-restart, this setup provides a *gap recovery* mode as outlined in 3.2.2.

Mapping a streams application graph to topologies of other supported execution engines, e.g. the Storm engine, allows for running applications under different semantics. As an example, the core elements *stream* and *process* directly map to Storm's *spout* and *bolt* elements which leads to an *at-least-once* execution semantic of the application on the Storm platform. Additionally, the execution incorporates Storm's high availability features.

### Scalability and Distribution

The streams framework itself does not provide automatic distribution among multiple machines when using the streams-runtime system. Graphs can however easily be split up

and manually started on different nodes while the runtime system provide auto-discovery of remote graphs and allows for accessing remote queues and services from distant compudation graphs.

When executing using the *Storm* engine, the data-flow graph is transformed into a Storm topology and inherits all the features provided by Storm. Any processes are wrapped into Storm bolts and can be executed among a distributed set of supervisor nodes of a Storm cluster.

### Usability and Process Modelling

A key objective of the streams framework is addressing the level of process design. The data-flow graphs are defined in XML using a small set of element. In addition, the XML directly incorporates streaming functions (written in Java) which allows for a high-level design of graphs, while keeping the gap between design and implementation as narrow as possible. This makes streams an *depdency injection* based framework for orchestrating streaming applications on a high-level using pre-compiled components.

By relying on Java's reflection API, new functions can be automatically detected if more libraries are added with no additional configuration required.

# 5 Summary

While the implementations of general stream processing frameworks presented in 3 do provide the abstract means of data processing in a continuous manner, they differ with regard to the guarantees they provide and the modelling capabilities they address.

A framework like streams does not provide any fault-tolerance features by its own runtime, but focuses on the abstract user-modelling of streaming applications using streaming functions. Storm provides fault-tolerance in a transaction save manner, but expects data to arrive in order, while MillWheel builds upon low watermark timestamps and does treat all data streams as unordered.

Looking at the history of development in each of these frameworks, their functionality has somewhat converged towards a common set of features, which are supported by either of the implementations:

1. Streaming applications defined as data flow graphs
2. Processing nodes/streaming functions as graph nodes
3. Partitioning of data streams for distribution
4. Restart of nodes and replay of data for fault-tolerance
5. Deduplication of messages for *exactly-once* semantics.

What makes a distinction even more difficult is the fact, that some of the functionalities inherent to one framework can easily be provided by additional user code within the other. Storm for example does not provide an API for state persistency, but includes callbacks for restoring state from external storage. This state persistency in contrast is directly integrated as part of the MillWheel API. Pushing this even further is S4, by providing automatic persistent checkpointing by the platform itself. However, this is tightly bound to user code, which may not be easily storable in an automatic fashion.

Storm itself did not provide support for Timers in versions earlier than `0.8.x`. These needed to be added with custom code, which has by now been superseeded by an inherent support for Timers.

This generally leads to a convergence of the various approaches to a set of frameworks that provide a joint set of fatures. As a result, frameworks that did not receive a reasonable acceptance, e.g. due to a too complicated usage or API, have been abandoned. Most prominent being the Yahoo! *S4* framework, which has been discontinued, as well as frameworks like *CIEL* or *Dryad*.

## 5.1 Comparison of Stream Processing Engines

In an attempt to extract the major benefits of each of the frameworks, we will give a walk-through of the features and highlight the capabilities of each framework for that feature.

### Message Processing Semantics

The basic semantic for processing messages is the number of times a message may be processed by a processing node. Typically all frameworks do support an *exactly-once* delivery by now. Earlier versions of *S4* were focusing on *at-most-once* execution, following the so-called *gap recovery* error handling, that accepts a (short) loss of messages.

The messages are in general ordered by their creation time and most framework pose a strict ordering of message while they are being passed between processing nodes. This ordering of messages is broken up in the partitioning of data streams where an ordering is only given on each partition of the stream. The *Storm* and *Samza* systems as well as the *streams-runtime* assume ordered message streams.

MillWheel [10] and OOP [37] are the only systems reviewed here, which inherently deal with out-of-order messages. Both systems are quite similar in that respect and use timestamps on messages with a low watermark that indicates the progress of work.

### State Handling and Fault Tolerance

Maintaining state is crucial for a number of stream processing tasks and is closely linked to fault tolerance handling. Even simple tasks such as counting elements require to manage the state of counters and possibly restoring these counts in case of a restart of the counter.

Despite the *streams-runtime*, which is geared towards embedded and single-node processing, all the frameworks provide fault tolerance by resending data that has not been acknowledged by a receiving node. Here, *Storm* provides an additional in-depth ack'ing by building a multi-node dependency chain that is required to be fully acknowledged by each node. However it does not integrate means for state persistency which needs to manually be handled by custom user code. The *streams-runtime* is following a gap-recovery approach using fail-fast and supervision. While this does not compare well to the sophisticated recovery mechanisms of the other frameworks, the targeting on single-node embeddable platforms makes this the most adequate way of dealing with outages.

*Samza* and *MillWheel* offer state management APIs that makes it easy to write user code which outsources any state handling to a backend, that is usually a itself a high available distributed storage system (Apache Cassandra, Spanner, BigTable, etc.). The streams API does provide a similar API for storing state in a process context which can be mapped to a distributed persistent store, but does only include a non-replicated in-memory store in its default embeddable *streams-runtime*.

### Scalability and Distribution

The scalability aspect is in princpiple handled by partitioning data streams based on some function and processing the partitions by multiple instances of the processing nodes. This core concept is provided by each of the frameworks in slightly different ways. Where *Storm* and *Samza/Kafka* address this by incorporating so called *groupings* or *partitions*,

the streams framework currently requires a more explicit manual layout of the partitioning and the data flow graph that corresponds to that.

A more declarative approach is provided by the *Stratosphere* system, which requires the user code to explicitly define properties of its input and output data in a way that allows for the provided compiler to create parallelized data flows from these descriptions. That property is unique to the *Stratosphere* approach.

Except for the *streams-runtime*, all the surveyed frameworks built upon a distributed execution environment, providing computation nodes that are managed by a central master node. Frameworks like *Storm*, *Samza* and *MillWheel* support a rebalancing of the executing processes among the computation nodes, allowing for a dynamic adaptation of the running data flow graph to changes in the data traffic distribution.

At the moment *MillWheel* seems to be the only framework that supports dynamic data flow graphs, which enable the users to add and remove processing nodes without restarting the system. This behavior is partly inherent to the publish-subscriber approach provided by *Samza*, as streaming applications are defined as loosely coupled components, connected by the queueing system.

The following Table 1 summarizes a features of the different stream processing frameworks. The support for some of the features is not always a binary distinction. As an example, the creation of dynamic graphs is not supported by Storm, but can be mimiced by dynamically adding new processing elements in a new topology that connects to the existing graph that is already exeuting.

| | | Apache Storm | Apache Samza | Yahoo! S4 | Google MillWheel | Stratosphere | CIEL | streams-runtime | streams + Storm |
|---|---|---|---|---|---|---|---|---|---|
| Messaging | exactly once | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ |
| | at most once | ✓ | | ✓ | ✓ | | ✓ | ✓ | ✓ |
| | at least once | ✓ | | | ✓ | | ✓ | | ✓ |
| | out of order | | | | ✓ | | | | |
| Persistent State | state-api | | ✓ | ✓ | ✓ | ? | | ✓ | ✓ |
| | state-custom | ✓ | ✓ | ✓ | ✓ | ✓ | ? | ✓ | ✓ |
| | state-auto | | | ✓ | | ? | ? | | |
| Scalability | partitioning | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| | distr. cluster | ✓ | ✓ | ✓ | ✓ | ✓ | | | ✓ |
| | rebalancing | ✓ | ✓ | ✓ | ✓ | | | | ✓ |
| Modeling | code level | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| | design level | | | ✓ | | | | ✓ | ✓ |
| | dynamic graphs | | (✓) | (✓) | ✓ | | ✓ | (✓) | |
| Embedded | embeddable | | | | | | | ✓ | |
| | android support | | | | | | | ✓ | |

Table 1: A feature matrix of the surveyed stream processing engines. Some properties (marked with "?") cannot be clearly validated due to missing documentation of the frameworks.

## 5.2 The Feature Radar

For a more global classification of the surveyed frameworks we look into a groaser categorization. With the more detailed distinctive features from Table 1, we can derive the higher-level landscape view as shown in the radar chart in Figure 22. Based on five categories, this figure roughly outlines the key aspects that each of the frameworks focuses on. Clearly, a few dominating aspects are the strong emphasis of *MillWheel* on the out-of-order processing and the streams framework aiming at process modelling and its focus on embedded setups.

The majority of the frameworks has been developed with the large scale processing requirements of *Big Data* in mind: being able to split the streams into partitions and to parallelize execution among a large set of distributed compute nodes.

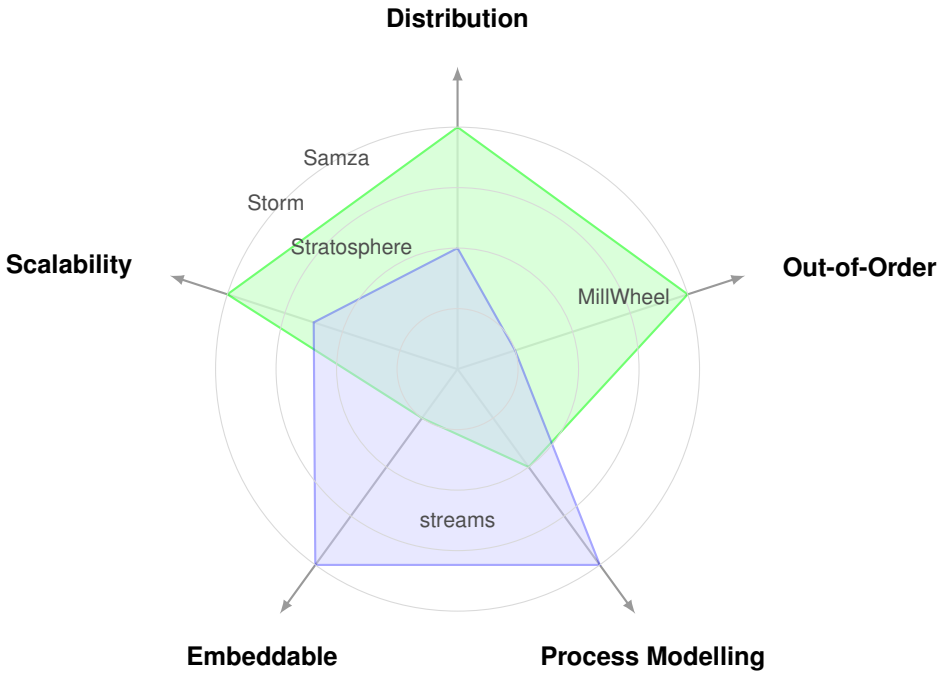



Figure 22: Key aspects which the general purpose streaming frameworks focus on. MillWheel is the sole framework additionally focussing on out-of-order processing, whereas streams is the only one geared towards embedded use. We only include the most active frameworks in that figure.

## 5.3 Comparing streams

A direct comparison of the *streams-runtime* with the other large scale distributed processing engines does not match well. The concept of streams is to provide a light-weight programming API and environment, that allows for the definition of data flow graphs without a direct dependency to a runtime engine. The *streams-runtime* provides a full

execution engine for such graphs with a very small footprint. This allows for running streaming applications on small devices like the Raspberry PI board or Android devices.

For managing larger data volumes, streams provides a mapping to Storm topologies, which enables users to define a streaming applications using the streams XML definitions and directly deploy these graphs on a Storm cluster. By moving the execution of such a streaming application from the *streams-runtime* to a Storm cluster, streams inherits a number of capabilities from the executing runtime environment.

Like the mapping of streams applications to Storm, a prototype for running the same applications on the *Samza* engine is currently under development.

The primary focus of streams is a high-level XML definition language that bridges the gap between data flow design and the integration of low-level Java classes directly into the design. By using Java reflection, custom code can directly be added into the data flow graph by referencing the custom classes as tags within the XML definition. With a number of generic *processor* classes already provided, users of streams can easily define streaming application without writing or compiling code.

The use of XML as a standardized definition language also supports graphical editors for application design to be used.

Albeit the benefit of defining streaming applications on a higher level (and independent of the underlying execution engine), streams serves as an *integration framework* that quickly allows for the incorporation of existing libraries directly into the streaming application. As an example, the MOA online learning library has been wrapped into streams , which allows for a number of popular learning algorithms to be used when designing an application.

### The streams Framework within SFB-876

As part of its focus of a resource-aware data analysis including the on-the-fly processing of streaming data, the streams framework has been adopted in the C3 project for handling large volumes of raw data of the FACT telesope as well as for the implementation and prototyping of data processing applications in projects A1 and B3.

The close coorporation of A1 with C3 has created a large amount of application specific streaming functions that allow for rapid prototyping of new feature extraction and processing pipelines by creating appropriate XML definitions. This makes streams an ideal environment for sharing processes among physicists and provides easy access to stream application modelling to students in that application domain.

## Acknowledgements

The streams framework has extensively been used in the ViSTA-TV project (EU grant number 296126) for user event processing and video feature extraction. This extensive use led to a number of improvements and bug fixes in the streams runtime.

# References


[1] Apache ActiveMQ.

[2] Spring framework. `http://www.springsource.org/`.

[3] Apache hadoop, 2007. `http://hadoop.apache.org/`.

[4] Apache zookeeper, 2008. `http://zookeeper.apache.org`.

[5] Samza, 2013. `http://samza.apache.org/`.

[6] Stratosphere, 2014.

[7] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Mitch Cherniack, Jeong hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Er Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *In CIDR*, pages 277–289, 2005.

[8] Charu C. Aggarwal. A survey of stream clustering algorithms. In *Data Clustering: Algorithms and Applications*, pages 231–258. 2013.

[9] Faruk Akgul. *ZeroMQ*. Packt Publishing, 2013.

[10] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[11] Ezilda Almeida, Carlos Abreu Ferreira, and João Gama. Learning model rules from high-speed data streams. In João Gama, Michael May, Nuno Cavalheiro Marques, Paulo Cortez, and Carlos Abreu Ferreira, editors, *UDM@IJCAI*, volume 1088 of *CEUR Workshop Proceedings*, page 10. CEUR-WS.org, 2013.

[12] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. The Royal Institute of Technology, Sweden, 2003.

[13] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 13–24, New York, NY, USA, 2005. ACM.

[14] Jürgen Beringer and Eyke Hüllermeier. Online clustering of parallel data streams. *Data and Knowledge Engineering*, 58(2):180 – 204, 2005.

[15] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Thomas R. Gabriel, Tobias Kötter, Thorsten Meinl, Peter Ohl, Kilian Thiel, and Bernd Wiswedel. Knime - the konstanz information miner: Version 2.0 and beyond. *SIGKDD Explor. Newsl.*, 11(1):26–31, November 2009.

[16] Albert Bifet. *Adaptive Stream Mining: Pattern Learning and Mining from Evolving Data Streams*, volume 207 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2010.

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

[18] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *In 2006 SIAM Conference on Data Mining*, pages 328–339, 2006.

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[20] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[21] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In Michel X. Goemans, editor, *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 30–39, 2003.

[22] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[23] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, April 2005.

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

[25] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM.

[26] Avigdor Gal, Sarah Keren, Mor Sondak, Matthias Weidlich, Hendrik Blom, and Christian Bockermann. Techniball: DEBS 2013 grand challenge. 2013.

[27] Sumit Ganguly. Counting distinct items over update streams. *Theoretical Computer Science*, 378(3):211–222, June 2007.

[28] The Stream Group. Stream: The stanford stream data manager, 2003.

[29] Sudipto Guha and Andrew McGregor. Approximate quantiles and the order of the stream. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2006.

[30] Peter Haider, Ulf Brefeld, and Tobias Scheffer. Supervised clustering of streaming data for email batch detection. In *Proceedings of the ICML*, pages 345 – 352. ACM, 2007.

[31] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013.

[32] Jeong-Hyon Hwang, Magdalena Balazinska, Alex Rasin, Ugur Çetintemel, Michael Stonebraker, and Stanley B. Zdonik. High-availability algorithms for distributed stream processing. In Karl Aberer, Michael J. Franklin, and Shojiro Nishio, editors, *ICDE*, pages 779–790. IEEE Computer Society, 2005.

[33] Piotr Indyk and D. Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the 37th STOC*, pages 202–208, 2005.

[34] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[36] Douglas Laney. 3D data management: Controlling data volume, velocity, and variety. Technical report, META Group, February 2001.

[37] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: A new architecture for high-performance stream systems. *Proc. VLDB Endow.*, 1(1):274–288, August 2008.

[38] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 346–357. VLDB Endowment, 2002.

[39] Floyd Marinescu. Esper: High volume event stream processing and correlation in java. Online article, July 2006.

[40] Ingo Mierswa, Michael Wurst, Ralf Klinkenberg, Martin Scholz, and Timm Euler. YALE: Rapid Prototyping for Complex Data Mining Tasks. In Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos, editors, *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2006)*, pages 935–940, New York, USA, August 2006. ACM Press.

[41] 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.

[42] J. Russell and R. Cohn. *Rabbitmq*. Book on Demand, 2012.

[43] Sherif Sakr, Anna Liu, and Ayman G. Fayoumi. The family of mapreduce and large-scale data processing systems. *ACM Comput. Surv.*, 46(1):11:1–11:44, July 2013.

[44] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Record*, 34(4):42–47, 2005.

[45] Daniel Warneke and Odej Kao. Nephele: Efficient parallel data processing in the cloud. In *Proceedings of the 2Nd Workshop on Many-Task Computing on Grids and Supercomputers*, MTAGS '09, pages 8:1–8:10, New York, NY, USA, 2009. ACM.