Special Issue

Stefan Hagedorn, Philipp Götze, Omran Saleh, and Kai-Uwe Sattler*

Stream processing platforms for analyzing big dynamic data

DOI 10.1515/itit-2016-0001
Received January 8, 2016; accepted May 9, 2016

Abstract: Nowadays, data is produced in every aspect of our lives, leading to a massive amount of information generated every second. However, this vast amount is often too large to be stored and for many applications the information contained in these data streams is only useful when it is fresh. Batch processing platforms like Hadoop MapReduce do not fit these needs as they require to collect data on disk and process it repeatedly. Therefore, modern data processing engines combine the scalability of distributed architectures with the one-pass semantics of traditional stream engines. In this paper, we survey the current state of the art in scalable stream processing from a user perspective. We examine and describe their architecture, execution model, programming interface, and data analysis support as well as discuss the challenges and limitations of their APIs. In this connection, we introduce Piglet, an extended Pig Latin language and code generator that compiles (extended) Pig Latin code into programs for various data processing platforms. Thereby, we discuss the mapping to platform-specic concepts in order to provide a uniform view.

Keywords: Dynamic data, stream processing, data analysis.

ACM CCS: Information systems → Data management systems → Database management system engines → Online analytical processing engines

1 Introduction

Big data is often characterized by the three Vs for volume, variety, and velocity where velocity represents the challenge of real-time or online processing of data. In many applications either the amount of data is too huge to store them first and process later or the data are valid (or useful) only for a certain period of time. Batch processing paradigms such as promoted by the Hadoop platform require to process the data repeatedly to construct/update models and, therefore, are not well suited for this kind of applications where data arrive continuously.

The problem of continuous processing of data has been already addressed ten years ago by so-called data stream management systems (DSMSs). Pioneering systems like STREAM [5] or Aurora [2] have introduced not only appropriate execution models and window-based query operators but also SQL-like query languages for stream processing. In later works, further approaches for complex event processing (CEP) [11] and incremental and window-based data mining have been developed.

Driven by the wide acceptance of big data platforms like Hadoop and Spark as well as new requirements, the stream processing paradigm currently gains great attention again. In the last few years, several new platforms have been developed focusing on scalable and reliable processing of streams in large cluster environments. Examples are – among others – MillWheel [3], Storm[1] [29], Spark Streaming[2] [30] or Flink Streaming[3]. Such large-scale stream processing platforms open several new applications, e.g., in the area of social network analysis (e.g., Twitter trends), Internet of Things (sensor data-based environmental monitoring and forecasting) or for real-time fraud detection and homeland security. However, these applications pose several new challenges to stream processing:

– Processing long-running queries in large compute clusters requires to provide reliability and elasticity, meaning that the system has to deal with node failures by maintaining replicas and/or implementing recovery strategies and to allow adding or removing of

***Corresponding author: Kai-Uwe Sattler,** TU Ilmenau, Databases and Information Systems group, D-98693 Ilmenau, Germany, e-mail: kus@tu-ilmenau.de
Stefan Hagedorn, Philipp Götze, Omran Saleh: TU Ilmenau, Databases and Information Systems group, D-98693 Ilmenau, Germany

1 http://storm.apache.org
2 http://spark.apache.org/streaming
3 https://flink.apache.org

nodes dynamically in order to handle workload fluctuations. Both properties require mechanisms for efficient state migration of query operators.
- Large volumes of dynamic data – possibly arriving at high rates – can be handled only by partitioning the input streams. This is particularly challenging for order-based operations such as aggregation over sliding windows, time-based operators, or CEP.
- Implementing complex analytical pipelines often requires to use a combination of standard operators (filter, join, and aggregates) for gathering, preparing, and combining data, analytics operators like pattern detection, clustering, regression, etc., as well as specific user-defined operators. This results in dataflow pipelines which are difficult or even impossible to express in SQL-like queries.

In order to address these challenges scalable stream processing platforms as the systems mentioned above rely on Hadoop technologies such as the distributed file system HDFS or resource managers like YARN or Mesos. Furthermore, in contrast to the traditional stream processing engines (SPEs) and their commercial derivatives like InfoSphere Streams [6], they do not offer declarative query languages like CQL but provide language-integrated APIs or domain-specific languages (DSLs). Combined with programming languages like Scala, Java, or Python this allows to implement the often needed application-specific functionalities as user-defined functions in a rather easy way but requires more advanced programming skills from the data scientists using the platform to implement their analytics pipeline.

On top of scalable stream processing platforms hybrid architectural patterns have been proposed recently which aim at combining stream and batch processing in a uniform architecture. An example is the lambda architecture [20] which consists of three layers: a batch layer for storing the master dataset and precomputing batch views, a speed layer for processing data not yet handled in the batch layer (e.g. arrived since the last batch computation), and a serving layer providing a queryable database for these batch views and the incrementally computed results of the speed layer. Particularly, the speed layer can be implemented using a stream processing engine.

In this paper, we survey the current state of the art in scalable stream processing from a user perspective. We describe the most prominent platforms with respect to their architecture, execution model, programming interface, and support for big data analysis. Furthermore, we discuss the challenges and limitations of the current APIs and present Piglet – an enhanced Pig Latin language and compiler – for specifying and compiling analytical pipelines on scalable stream processing platforms.

2 Scalable stream processing platforms

Several stream processing solutions have been developed over the years. The solution space ranges from *(a)* low-level *publish-subscribe messaging middleware* like Apache Kafka over *(b)* general-purpose *data stream processing platforms* like Apache Storm, Spark Streaming, and Flink Streaming to *(c)* high-level *declarative DSMSs* like STREAM and Aurora or the commercial products InfoSphere Streams, Streambase [28], and SQLstream [27].

Currently, high scalability for big dynamic data is mainly addressed by approaches from group *(b)*. Therefore, we focus in the following on this group but refer also to related work in traditional DSMS and CEP systems of group *(c)* at the end of this section.

Although every framework provides scalability and fault tolerance, they differ fundamentally in their processing model, architecture, and execution model. In general, there are two main different paradigms to process a live data stream: (1) tuple-wise (true stream processing) is to process the data tuple-by-tuple as it arrives from sources and (2) micro-batching is to divide the stream into chunks of specific time units (called "micro-batches") and processing them individually. A summary of the characteristics of the SPEs, which are described in the following, is given in Table 1.

Apache Storm. The first widely used open-source scalable stream processing platform was Apache Storm. It has been presented by Twitter [29], supports fault-tolerance, and offers a strong guarantee that every data item (i.e., tuple) will be fully processed at least once. Broadly, Storm processes streams on a tuple basis. Although it is written primarily in Clojure and Java, its tasks and processing components can be defined in any language by implementing a simple communication protocol.

Storm provides stateless processing by introducing `spout` and `bolt` components, where the former is a source of tuples pulled from, e.g., message queues, whereas the latter consumes input streams, performs some processing and transformations, and passes them downstream to subsequent bolts or external systems. Wiring spouts and bolts together leads to a directed graph, called a `topology`.

To achieve fault-tolerance and "at least once" guarantee in Storm, the directed acyclic graph (DAG) of each tuple

Table 1: Summary of all described stream processing engines [13, 19, 29, 31, 32].

	Storm (Core)	Spark Streaming	Flink Streaming
API	Low-Level	High-Level	High-Level
Language Support	Any	Java, Scala, Python	Java, Scala, Python
Windowing	Not built-in	Only Time-based	Time-based, Row-based, Data-driven with some limitations
Resource Managers	YARN, Mesos, Built-in	YARN, Mesos, Built-in	YARN, Built-in
Computation Model	Tuple-wise	Micro-batch	Tuple-wise
Latency due to Model	Low	Medium	Low
Stream Primitive	Tuple	DStream	DataStream
Fault Tolerance Mechanism	Record ACKs	Micro-batching	Distributed Snapshots
Guarantee	At least once	Exactly once	Exactly once
Stateful Operations	No	Yes	Yes
Flow control	Problematic	Problematic	Natural

emitted by a spout is tracked by special `acker` bolts/tasks. For that, the responsible spout assigns an ID to each new tuple. In the case that a tuple failed to be completed (i.e., a particular bolt indicates the tuple has failed or it exceeds its timeout to process), Storm uses the tuple's ID to replay it from the spout that created it. For the positive case that a tuple is completed in the whole topology, the responsible acker task sends the final ack to the spout that emitted the tuple, which can then remove it from the queue. Each spout or bolt is associated with a set of tasks, which in turn are executed by several instances (called executors or threads) to allow a parallel computation. The parallelism degree for each task can be specified when building the topology. Based on the topology configuration, Storm distributes the work evenly to nodes in the cluster. Therefore, each node executes a subset of all the tasks for the topology. To manage its cluster resources, Storm has its own scheduler or can be run on top of YARN or Mesos. ZeroMQ and/or Netty middlewares are used for inter-node communication when a task of one machine needs to send data to a task of another machine.

There are two types of nodes in the Storm cluster, namely the master node and the worker nodes. The first runs a daemon called `Nimbus` responsible for distributing and deploying code around the cluster (i.e., after packaging up a JAR file containing the Storm topology and all of its dependencies) and monitoring failures in the system. A `Supervisor`, which listens for a task assigned to it by Nimbus, runs on each worker node. The communication between Nimbus and the worker nodes as well as the cluster state is managed by ZooKeeper[4]. To integrate Storm with external technologies it provides specific modules for Apache Kafka, HDFS, HBase, Hive, Redis, etc.

Spark Streaming. Spark represents a platform that natively supports both batch and streaming tasks. It is written in Scala and provides specific API support for Scala, Java, and Python. The base engine is Spark core which is responsible for most of the tasks such as memory management, deploying and monitoring tasks on a cluster, failure handling, and interacting with storage systems. The basic core abstraction for data elements in Spark is called Resilient Distributed Dataset (RDD). An RDD simply represents an immutable, fault-tolerant, and distributed dataset. Each RDD is split into multiple partitions, which can be computed on different nodes of the cluster [30].

Spark Streaming is an extension of the core Spark API (i.e., higher-level library) that enables scalable processing and sophisticated analytics of live data streams. It leverages the Spark's core computation strategies to perform stream processing. To maintain compatibility with Spark's core and its resilient characteristics, it uses the `DStream` [31, 32] data structure which is basically a series of RDDs, to process the stream data. It treats each batch of incoming data as a discretized stream (*micro-batch*) and processes them using RDD operations of the core API. Thus, a streaming computation represents a series of deterministic batch computations of a definable time interval size. This approach naturally leads to a higher latency than for true streaming engines, but in exchange is also more robust to node failures and provides exactly once semantics. It is achieved by tracking a graph of deterministic operations (*lineage*) in each DStream and in the case of failure the partitions are recomputed by rerunning the tasks that produced them. On top of that, the system periodically cre-

4 https://zookeeper.apache.org/

ates checkpoints to avoid too much recomputation. The micro-batch architecture also has problems with backpressure if for instance the sink is slower than the source operation, which either leads to more or bigger subsequent batches. However, since version 1.5, Spark offers a backpressure option, which automatically sets the rate limits.

In Spark Streaming both stateless and stateful operators can be applied to DStreams. Stateless means that the operation can be applied independently for each incoming batch such as `map`. Stateful, on the other hand, requires a state data type and a corresponding update function, e.g., for the mean value by maintaining the sum and count of the determined field as data structure and the quotient of them as function. In general, Spark runs on top of HDFS and has the ability to support existing resource management systems such as YARN and Mesos for deployment and scheduling purposes. In addition to these managers, it can run as a stand-alone cluster using its built-in scheduler. For interconnecting with external systems, Spark provides connectors for inter alia Apache Kafka, Flume, Twitter, ZeroMQ and Kinesis.

Flink Streaming. According to [13] and the official documentation[5], Flink Streaming is designed to process big data as fast as possible with low data latency (i.e., can be adjusted dynamically) and high fault tolerance on distributed nodes. The basic concept to achieve the low latency is pipelining, i.e., applications are organized as dataflows with continuous operators. Thereby, they enable a natural flow control, in which, e.g., slow data sinks backpressure rapid sources. Furthermore, the tuples are collected in buffers with an adjustable timeout before they are sent to the next operator to turn the knob between throughput and latency. For instance, when setting the timeout to zero, tuples are directly forwarded in order to achieve a latency-optimized flow.

There are a few similarities between Spark Streaming and Flink Streaming including their operators, components, supporting both batch and stream processing as well as their capabilities of running in a stand-alone mode and on YARN. For interfacing with other systems, Flink Streaming provides connectors for Apache Kafka, Elasticsearch, HDFS, RabbitMQ, Twitter Streaming API and also allows to define custom connectors. In contrast to Spark Streaming, Flink Streaming primarily processes the data stream in a tuple-wise manner.

Flink is mainly written in Java and has API support for Scala, Java, and Python. The APIs are built upon a distributed streaming dataflow engine, whereby the batch variant is treated as a special case of streaming applications to run more efficiently. In Flink Streaming transformations and modifications are applied to `DataStream` objects, which are the basic data abstraction. Most of the operators take a data stream as input and provide a new data stream as output. When using stateful computations, like it is done for window buffers, Flink ensures exactly once semantics. To achieve this fault tolerance, it uses a mechanism called Asynchronous Barrier Snapshotting (ABS) [9], which is a lightweight approach to determine the global states of distributed systems. It is based on distributed snapshots originally presented by Chandy and Lamport [10]. The implementation makes use of *stream barriers* moving through the dataflow alongside with the data. Whenever a barrier arrives at an operator, it aligns the flow until it got the barriers with the same ID from all input streams and divides the data in records before and after the barrier. Once all barriers have arrived at the operator the current state is reported to a state backend (e.g., HDFS) and the barrier is emitted into the output streams. These snapshots are initiated by the master via sending checkpoint messages to all source nodes, which report the current offset of the input stream. In the case of a failure, the last complete snapshot is used to reset all the nodes.

Traditional distributed DSMS and CEP. Distributed processing of streams, partitioning, and fault tolerance have been also studied for traditional DSM and CEP systems. For example, Borealis [1] introduces replicated processing nodes as well as several new tuple types such as punctuation tuples and control tuples like undo tentative and done tuples. Tentative tuples are tuples resulting from processing a subset of the input which can be corrected later and done tuples indicate the end of state reconciliation, i.e. the process of stabilizing the output. Thus, this approach aims mainly at fault-tolerance but not at partitioning. Another approach is Telegraph's FluX [26] which addresses the load balancing problem by partitioning while providing fault tolerance for pipelined dataflows. FluX extends the idea of the exchange operator from parallel query processing and encapsulates state partitioning and tuple routing for repartitioning stateful operators even during query execution. Partitioning functions particularly for stateful stream operators are also discussed e.g. in [33] and [15]. An auto-parallelization technique to implement elastic scaling is presented in [16].

The specific problem of distributed and/or parallel CEP is addressed by different approaches. A rewriting-based approach for distributing event automata across a cluster of nodes is presented in [25]. Partitioning tech-

5 https://ci.apache.org/projects/flink/flink-docs-master/

niques for CEP are described e.g. in [8], [18] and [23]. Strategies for deploying (i.e. load distribution and operator placement) CEP queries are described in [12].

3 APIs and DSLs for stream processing and analytics

Platform solutions as discussed as group *(b)* in the previous section provide mainly a programming interface for JVM-based languages (Java, Scala) as well as Python. For this purpose they offer abstractions of distributed datasets (`RDD` as low level abstraction, `DataFrame` and `Dataset` as higher-level abstraction on Spark; `DataStream`, `DataSet`, and `Table` in Flink) on which operators are defined. Among these operators there are so-called transformations, which transform an immutable dataset to an output dataset and actions computing a result and return it back to the driver program. Because datasets and streams are horizontally partitioned these operators are parallelized automatically, i.e., as in MapReduce the data parallelism is transparent to the user. Standard transformations are `map` to apply a function to each tuple, `filter`, set operations like `union` or `intersect`, `distinct` as well as `grouping`, `sorting`, and `join` where the latter three transformations require the definition of a key field and involve a `shuffle` operator for redistributing the data among the partitions according to the key value.

Particularly, the functional programming features of languages like Scala allow a very comfortable definition of sequences of transformations into dataflow programs. Data stream abstractions are exposed as generic classes as the following Flink example shows:

```
val inStream: DataStream[String, Int, Double] = ...
val result = inStream
  .filter { case (_, i, _) => i > 100 }
  .map { case (s, _, d) => (s, d + 10) }
```

Batch processing or traditional database systems know exactly when the evaluation of data is finished: when all entries in the file or table have been processed. Because live stream data never ends, any stream processing engine must support window operators to restrict the number of tuples to be included in a computation. In general, a window operator defines the "scope" of data for streaming operators such as aggregation or joins. Windows can be time-based (i.e., defined over time) which is probably the most common window, or row-based (i.e., depends on the number of tuples). Such windows should also be able to slide at a particular amount from the current window (e.g., a window could have a size of five tuples and the next window could slide by one tuple from the current one). As a result, depending on window size and slide size values, windows can be disjoint or overlapped. Flink Streaming and Spark Streaming both support window operators. Flink Streaming provides a number of pre-defined windows for keyed and non-keyed data streams. The former is a window that contains elements with the same key while the latter is a non-parallel window containing different elements, thus, it will be evaluated at a single node. Moreover, Flink Streaming supports both types of windows, time-based and row-based windows. An example of a non-parallel window is the following:

```
env.fromCollection(tuples)
      .timeWindowAll(Time.of(1, TimeUnit.SECONDS))
      .sum(1).print()
```

Here, `timeWindowAll` represents a time-based non-keyed window. Let's illustrate the window usage with a join operator in the following example. We suppose to have two data streams, `streamA` with the schema `(id: Int, value: Int)` and `streamB` with a schema `(id: Int, name: String)`, and we want to join these data streams together to get a complete record. Since we are dealing with stream data, a window operator should be used.

```
val joined = streamA.join(streamB).where(_.id)
      .equalTo(_.id).countWindow(1000, 100)
      .apply{ (a,b) => Record(a.id, b.name, a.value)}
```

Here, `countWindow` represents a count-based window of 1000 tuples, that "slides" every 100 tuples. One of the major restrictions in Flink Streaming is that there is no possibility to join two already windowed data streams together (i.e., different sizes for windows).

In contrast to Flink Streaming, Spark Streaming supports only a generic time-based window. Since Spark Streaming handles streams as micro-batches, the parameters of its window (i.e, window size and slide interval values) must be multiples of the batch interval of the source DStream. The following example counts the number of times each hashtag has occurred in a window of size 1 min and it slides every 5 s.

```
val ssc = new StreamingContext(sparkContext,
                                        Seconds(1))
val tweets = TwitterUtils.createStream(ssc, auth)
val hashTags = tweets.flatMap (tweet =>
                                  extractTags(tweet))
val counts = hashTags.window(Minutes(1),
                           Seconds(5)).countByValue()
counts.foreachRDD(rdd => rdd.foreach(
                              elem => println(elem)))
```

The Java and Python APIs provide similar features but are a bit more verbose.

Compared to the Spark and Flink APIs, Storm provides only a low-level API defining mainly a `TopologyBuilder` class for constructing the directed graph of operators as well as interfaces for spouts and bolts. The actual transformations of bolts – which can process multiple input streams and produce multiple output streams – have to be implemented by the user. The Trident[6] library (see below) provides a set of standard operators similar to the transformations in Spark and Flink as well as functionality to simplify the construction of Topologies. Currently, Storm itself does not have a direct implementation of window operators.

To overcome Storm's limitations, Trident provides various stateful operators like group-by, aggregate, join, filter, etc. for Storm. It provides an abstraction for maintaining the state of these operators either internally (memory) or by storing this state in external sources (e.g., Memcached or Cassandra) and supports exactly-once processing semantics. This abstraction allows the developer to focus on the problem rather than focusing on the construction of pure bolts and spouts for a topology. While Storm processes streams on per tuple basis, Trident processes streams in micro-batches in order to reduce the number of times it has to communicate with the persistent storage for fault tolerance, because the entire batch may be committed as an atomic transaction. Using batch instead of individual tuple processing somewhat increases latency but improves overall throughput. In Trident, spouts remain conceptually the same as in Storm, but are implemented in a different way. They consist of one Storm spout and one or more bolts to handle batch and stateful operations correctly. In contrast to Storm, developers do not use bolts in Trident anymore. These bolts are substituted with higher level semantics of filters, functions, aggregates, joins, merges, and states. Developers can also define their own custom operators – if needed. Trident then takes its topology and turns it into an efficient Storm topology for execution on a cluster.

To support windows, the authors of Trident suggest to use `partitionPersist` and `stateQuery` functions provided by the API.

New libraries combine stream processing engines with machine learning (ML) functionality and provide APIs and DSLs to users and developers. Their collection of operators, however, is not as comprehensive as, e.g., in R or Weka for batch processing on a single machine. The Mahout[7] library provides a comprehensive collection of ML operators for the Hadoop MapReduce platform, but does not offer operators to work with data streams. As Spark and Flink are more and more replacing MapReduce installations, ML libraries have been developed for these platforms, too. Spark MLlib[8] provides several types and algorithms for batch operations, but only one operator for streaming data: a k-means clustering operator. Similarly, the FlinkML project aims to provide such operators for the Flink platform and also to exploit the streaming functionality. However, at the current state only a few operators are available[9]. To define and identify complex events the FlinkCEP[10] project can be used.

Two examples for comprehensive ML libraries for stream engines are Trident-ML[11] for Storm/Trident and Apache SAMOA[12]. For Storm the (seemingly abandoned) Trident-ML library provides ML operators, like clustering, stream statistics, linear regression, etc. SAMOA is a platform and library for ML tasks on data streams [7, 14]. As MOA is known for ML operators for non-distributed stream processing, SAMOA provides operators to execute ML tasks in a distributed streaming environment. Programs for SAMOA can be executed in the distributed SPEs Storm, S4 [21], and Samza[13]. These different platforms are encapsulated by an extra SAMOA-SPE layer, which contains the necessary adapters to communicate with the respective engine. This way, SAMOA hides the specifics of each platform, making the underlying SPE transparent to the user.

6 http://storm.apache.org/releases/2.0.0-SNAPSHOT/Trident-tutorial.html

7 https://mahout.apache.org/

8 https://spark.apache.org/docs/latest/mllib-guide.html

9 https://ci.apache.org/projects/flink/flink-docs-master/apis/batch/libs/ml/index.html

10 https://ci.apache.org/projects/flink/flink-docs-master/apis/streaming/libs/cep.html

11 https://github.com/pmerienne/trident-ml

12 https://samoa.incubator.apache.org/

13 http://samza.apache.org

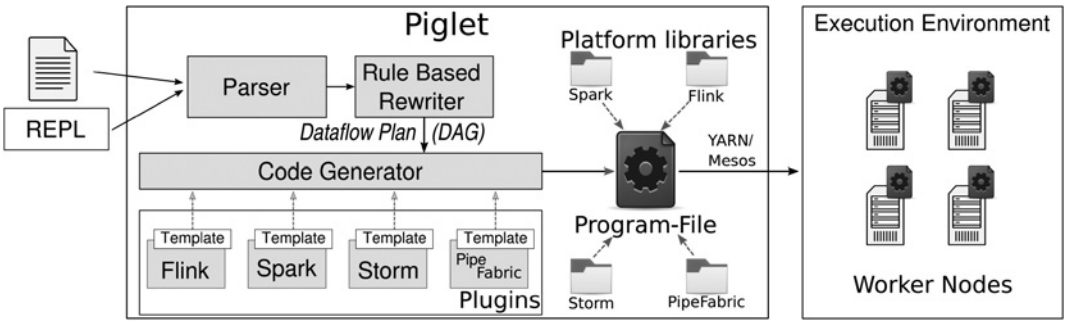

Figure 1: Piglet's internal architecture: The code generator uses the backend plugins to create platform specific programs.

Users write their programs using the SAMOA DSL for Java. The library part contains a collection of ML operations for, among others, classification, clustering, and regression. For classification, SAMOA provides a Vertical Hoeffding Tree, where attributes of an item are processed in parallel, instead of processing multiple items in parallel. To compute clusters in the distributed environments, SAMOA includes a distributed implementation of Clu-Stream. This implementation computes micro-clusters online which are refined by a batch process that is triggered periodically. Adaptive Model Rules can be used for regression, where SAMOA uses both, horizontal and vertical parallelism to distribute the rules in the cluster. Next to these built-in ML operators, SAMOA allows to add additional libraries via a ML adapter.

4 A dataflow language for stream processing

The platforms and engines introduced in the previous sections all provide their own APIs for defining analysis pipelines. Users have to implement a program in a programming language supported by the platform, compile, and deploy it to the execution environment. This requires the user to learn a programming language and the API of the chosen platform. Language-integrated DSLs mitigate this problem and provide a flexible way to express all kinds of tasks. However, this tight integration into programming languages makes it difficult to exploit optimization techniques known from query processing such as reordering of operators or algebraic rewriting. Additionally, analysis pipelines written for one platform cannot easily be ported to another platform, but users would have to rewrite all their code and adapt it to the new API.

In contrast, declarative query languages like CQL offer an abstraction over the underlying implementation of the specific operations and allow the optimization of queries by exploiting algebraic rules and cost models to create efficient internal execution plans. Their possibilities to express complex workflows, however, are limited and queries quickly become impossible or too complex to express. Such tasks are for example iterative operations or the integration of user-defined functions.

Dataflow languages like Pig Latin [22] have emerged not only in the context of Hadoop as flexible and powerful tools for implementing complex analysis tasks. These languages provide schema flexibility, support common programming patterns such as iteration as well as extensibility through user-defined functions while avoiding the shortcomings of classical query languages.

For this reasons we have adopted Pig Latin as a dataflow language and extended it for stream processing. The result called Piglet[14] is a compiler and code generator that processes (extended) Pig Latin scripts, optimizes them, and generates programs for various backend platforms. Currently, Piglet supports both the streaming platforms Spark Streaming, Flink Streaming, Storm, and our own C++-based framework PipeFabric [24] as well as the batch processing platforms Spark and Flink and the traditional Hadoop-based Pig engine as backends. Figure 1 shows an architectural overview of Piglet. The differences of the various backends are hidden by the platform libraries and the code generator templates.

By providing a unified model of dataflows for different stream and batch processing platforms, Piglet pursues similar goals like the Google Dataflow model [4] or the recently incubated Apache Beam[15] project. However, Apache Beam relies on a (Java-based) language-specific DSL whereas Piglet provides a high-level dataflow language. Nevertheless, Google Dataflow could be used as

14 https://github.com/ksattler/piglet
15 http://incubator.apache.org/projects/beam.html

an additional backend for Piglet and Apache Beam can simplify the code generation for different backends in the future.

In the following, we will describe Piglet's features and present the stream processing extensions.

Window & CEP operators. Our Pig Latin language extensions for stream processing are based on our previous work [23] and comprise the following classes of operators:
- streaming source and sink operators,
- windows,
- complex event processing (CEP).

The source and sink operators are used to read and write data from and to network streams. These `SOCKET_READ` and `SOCKET_WRITE` operators are able to connect to plain TCP or UDP streams, but can also communicate via ZeroMQ and Kafka.

Several classes of windows are typically implemented in SPEs: time- vs. row-based windows, sliding, tumbling, landmark or even tilted windows. Currently, Piglet supports both time- and row-based windows but relies on the actual implementation of the underlying stream processing platform. Windows are defined by a separate dataflow statement, `WINDOW`, allowing to apply multiple sequences or graphs of dataflow operators to a window stream. However, for platforms like Flink Streaming where operators such as join require an explicit window this requires to determine the scope of the window automatically. This is done in the rewriting phase of the Piglet compiler with dedicated rewriting rules. The following example shows a simple word count script, where the frequency of words is computed over a sliding window of 5 min:

```
in = SOCKET_READ '127.0.0.1:8889'
      AS (line: CHARARRAY);
words = FOREACH in
      GENERATE flatten(tokenize(line)) AS word;
w = WINDOW words RANGE 300 SECONDS;
grpd = GROUP w BY word;
cntd = FOREACH grpd GENERATE group, COUNT(w);
```

When working with streams, one often needs to identify recurring patterns within the input stream data which is done using CEP operations. Here, an event is an incoming data item (tuple) and sequences and combinations of such single events form a complex event. Logical operators like conjunction, disjunction, and negation as well as temporal operators, e.g., sequence, are used to express how the single events must be related to each other in order to form a complex event [11]. Event pattern matching is supported in Piglet by the `MATCH_EVENT` operator. It allows to define the aforementioned conditions together with the `WINDOW` operator to specify the time or range window of validity. The following example illustrates the usage of this operator to find all sequences of events (i.e., vehicles) that are in the same road segment and drive in the same direction within a time window of 60 s.

```
a = SOCKET_READ '127.0.0.1:8889' AS (ts: long,
      id: int, speed double, dir: int, Seg: int);
b = WINDOW a RANGE 60 seconds;
c = MATCH_EVENT b PATTERN SEQ (A, B) WITH (B:
      (id == A.id && dir == A.dir && Seg == A.Seg));
```

Piglet's `MATCH_EVENT` operator is mapped to platform-specific operator implementations which use non-deterministic finite automata (NFA) to evaluate event patterns over the platform-specific stream abstractions, e.g., `DStream` in Spark Streaming.

Rewriting & optimization. The input is parsed into a DAG of logical operators, called a dataflow plan. Then, the rule-based rewriter is applied to the plan. Our currently implemented rule set includes well-known operations like pushing down selections to the source but also transformations that produce native Pig statements for our extensions.

As an example, consider Linked (Streaming) Data. For querying Linked Data, Basic Graph Patterns (BGP) are a core concept. A BGP implicitly contains join and filter operations that are executed while mapping it to the underlying Linked Data stream. To support BGPs, we introduced a new `BGP_FILTER` operator, that allows to include BGPs into a Pig script. Internally, when the rewriter finds such an operator, it analyzes the content of the BGP and transforms it into a sequence of join filter combinations based on pre-defined rules [17].

A second rewriting case is the processing of `WINDOW` statements. For instance, complex event patterns are typically detected within a certain time window. Thus, the `MATCH_EVENT` operator should be combined with a `WINDOW` previously defined on the input data stream. Another example is the code generation for window joins in Flink Streaming which are implemented as dedicated join operators requiring an explicit window size parameter. All these cases are handled in Piglet with rewriting rules.

Backends & code generation. Piglet is able to generate programs for various target platforms, called backends, from a single Pig Latin script. A backend is a library plugin for Piglet containing necessary files and functions to generate the target program and to submit the program to

the specific platform. As shown in Figure 1, the Pig Latin input is passed to the code generator which uses the backend plugins to create the program. This input can be a list of files or entered interactively via a Read-Eval-Print-Loop (REPL) shell similar to Spark Shell or Pig's Grunt.

The compiled program is then combined with a platform-specific Piglet library in a *jar* file and finally deployed to the execution environment via YARN or Mesos. The platform-specific library provides abstractions to hide implementation details, e.g., for the source, sink, and CEP operators. Since these backends are only plugins, the compiler can be easily extended to support other platforms by creating a plugin implementing some interfaces and providing a template file that the code generator will use to create the backend dependent code.

APIs and language-integrated DSLs as discussed in Sect. 3 require to use generic field accessors (e.g., `_1`, `_2` in Scala APIs) or the create application-specific classes for representing records. Though, this problem can be alleviated when using Scala's case classes or recent APIs such as SparkSQL's `DataFrame`s, it still requires some boilerplate code for representing, wrapping, and unwrapping data records and field values. Piglet creates such code automatically as schema classes providing typesafe fields, accessors, and serialize/deserialize methods.

Consider the Pig Latin script for Piglet that connects to a stream, filters the incoming data on the first column, and simply prints the results on the screen:

```
a = SOCKET_READ '127.0.0.1:8889'
      AS (x: int, y: double);
b = FILTER a BY x > 10;
DUMP b;
```

For Flink Streaming, Piglet will create the following Scala code, along with some helper functions:

```
case class _t1_Tuple (_0: Int, _1: Double) ...

object TestApp {
 def main(args: Array[String]) {
  val env =
   StreamExecutionEnvironment.getExecutionEnvironment
  val a = PigStream[_t1_Tuple]()
            .connect(env, "127.0.0.1", 8889)
  val b = a.filter(t => t._0 > 10)
  b.map(_.mkString()).print
  env.execute("Starting Query")
 }
}
```

Here, `PigStream` is a custom loader function that is automatically included and `_t1_Tuple` is a helper class representing the current schema of the stream.

UDF. In the traditional Pig engine, user defined functions (UDFs) are included by compiling them into a *jar* file that is added in the Pig Latin script via the `REGISTER` statement. This is also supported by Piglet, but we provide an even simpler way to include such UDFs: Users can directly embed their functions inside <% .. %> within the Pig script. The code marked by these tags is directly integrated into the generated program. This way, users do not have to perform the tedious task of creating a separate *jar* file for only a few small functions.

5 Conclusion

Emerging applications, e.g., in the areas of cyber-physical systems, IoT or social network analysis have increased the demand for approaches and technologies for online or real-time analysis of big dynamic data. The combination of established data stream and complex event processing techniques with highly scalable and reliable distributed platforms promises solutions capable to address these challenges. In this paper, we have presented a survey of the current state of the art of these platforms by particularly focusing on architectural and API aspects. Furthermore, we have discussed a dataflow language and compiler as extension of the well-known Pig Latin language which aims at integrating stream processing and CEP language features while supporting multiple target platforms.

References

1. D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.
2. D. J. Abadi, D. Carney, U. Çetintemel, and et al. Aurora: a new model and architecture for data stream management. *VLDBJ*, 12(2):120–139, 2003.
3. T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.
4. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

5. A. Arasu, B. Babcock, S. Babu, and et al. STREAM: The Stanford Data Stream Management System. *Book chapter*, (2004-20), 2004.
6. A. Biem, E. Bouillet, H. Feng, and et al. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *SIGMOD*, 2010.
7. A. Bifet and G. De Francisci Morales. Big Data Stream Learning with SAMOA. In *ICDMW*, 2014.
8. L. Brenna, J. Gehrke, M. Hong, and D. Johansen. Distributed event stream processing with non-deterministic finite automata. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 3:1–3:12, New York, NY, USA, 2009. ACM.
9. P. Carbone, G. Fóra, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight Asynchronous Snapshots for Distributed Dataflows. *arXiv preprint arXiv:1506.08603*, 2015.
10. K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *TOCS*, 3(1):63–75, 1985.
11. G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.
12. G. Cugola and A. Margara. Deployment strategies for distributed complex event processing. *Computing*, 95(2):129–156, 2013.
13. Data Artisans. High-throughput, low-latency, and exactly-once stream processing with Apache Flink. http://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/.
14. G. De Francisci Morales. Samoa: A platform for mining big data streams. In *WWW*, pages 777–778, 2013.
15. B. Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539, Aug. 2014.
16. B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, June 2014.
17. S. Hagedorn, K. Hose, and K.-U. Sattler. SPARQling Pig – Processing Linked Data with Pig Latin. In *BTW*, 2015.
18. M. Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, New York, NY, USA, 2012. ACM.
19. S. Landset, T. M. Khoshgoftaar, A. N. Richter, and T. Hasanin. A survey of open source tools for machine learning with big data in the Hadoop ecosystem. *Journal of Big Data*, 2(1):1–36, 2015.
20. N. Marz. Big Data Lambda Architecture. http://www.databasetube.com/database/big-data-lambda-architecture/, 2012.
21. L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed Stream Computing Platform. In *ICDMW*, pages 170–177, 2010.
22. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.
23. O. Saleh, S. Hagedorn, and K.-U. Sattler. Complex Event Processing on Linked Stream Data. *Datenbank-Spektrum*, 15(2):119–129, 2015.
24. O. Saleh and K.-U. Sattler. DEBS Grand Challenge: The PipeFlow Approach. In *DEBS*, pages 326–327, 2015.
25. N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.
26. M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.
27. SQLstream. Real-time Processing for Big Data in Motion. http://www.sqlstream.com.
28. TIBCO Software. StreamBase. www.streambase.com.
29. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.
30. M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.
31. M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.
32. M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.
33. E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. *PVLDB*, 4(11):1181–1188, 2011.

Bionotes

Stefan Hagedorn
TU Ilmenau, Databases and Information Systems group,
D-98693 Ilmenau, Germany
stefan.hagedorn@tu-ilmenau.de

Stefan Hagedorn studied Computer Science at the TU Ilmenau and received his Diploma (M.Sc.) in 2010. Since 2012 he has been a member of the Databases and Information Systems group at the TU Ilmenau, Germany. His current research focuses on Big Spatio-temporal Data and optimizing dataflow programs.

Philipp Götze
TU Ilmenau, Databases and Information Systems group,
D-98693 Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Philipp Götze received his bachelor degree in 2013 from the Cooperative State University, Mannheim, Germany. In 2015 he received

his M.Sc. from the TU Ilmenau, Germany. In 2016 he joined the Databases and Information Systems group at the TU Ilmenau. His research interests include optimizing dataflow programs as well as stream processing.

Omran Saleh
TU Ilmenau, Databases and Information Systems group,
D-98693 Ilmenau, Germany
omran.saleh@tu-ilmenau.de

Omran Saleh received his bachelor degree in 2010 from Birzeit Univeristy, Palestine. In 2012 he received his M.Sc. from the TU Ilmenau, Germany and then joined the Databases and Information Systems group. His research topics include stream processing and Complex Event Processing.

Prof. Dr. Kai-Uwe Sattler
TU Ilmenau, Databases and Information Systems group,
D-98693 Ilmenau, Germany
kus@tu-ilmenau.de

Prof. Dr. Kai-Uwe Sattler leads the Database and Information Systems group at the Faculty of Computer Science and Automation of the Ilmenau University of Technology, Germany.