CHAPTER

# A TAXONOMY AND SURVEY OF STREAM PROCESSING SYSTEMS

11

**Xinwei Zhao*, Saurabh Garg*, Carlos Queiroz†, Rajkumar Buyya‡**

*School of Engineering and ICT, University of Tasmania, Tasmania, Australia †The Association of Computing Machinery (ACM), Singapore ‡Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

## 11.1 INTRODUCTION

Every second an incredible amount of data is being generated around the world. According to [13], the world has created 90% of all its data in the last two years. Such staggering number indicates that the data nowadays is not only large in its size but also implies the fast speed in its generation and changes. The amount of data generated is so large that it is not only difficult to manage but also process using traditional data management tools. This data is referred as big data in today's context. There are two fundamental aspects that have changed the way data needs to be processed and thus needs a complete paradigm shift: the size of data has evolved to the amount that it has become intractable by existing data management systems, and the rate of change in data is so rapid that processing also needs to be in real-time. To tackle the challenge of processing big data, MapReduce framework was developed by Google, which needs to process millions of webpages every second. Over the last five years with the availability of several open-source implementations of this framework such as Hadoop [34], it has become a dominant framework for solving big data processing problems. However, recently it was observed that "Volume" is not the only challenge of big data processing and that the speed of data generation is also an important challenge that needs to be tackled for processing sensing data which is continuously being generated [8]. There are a number of applications where data is expected to be ingested not only in high volume but also with high speed, requiring real-time analytics or data processing. These applications include: stock trading, network monitoring, social media based business analytics, and so on. More specifically, there is a high demand of real-time data processing on the Internet or in sensors-related business models as the data generated need to be analyzed dynamically. For example, Google needs to count the clicks of websites in real time to decide which webpages are popular, and then use this information to leverage the advertisement fees to earn benefits. Besides, there is also a value associated with each dataset that varies with time. For example, static pages may have validity/value for some months; blogging may have for days, and Twitter messages may be valuable for less than a day. To process and analyze a data set which is changing in its value/validity quite fast, it is not productive to apply traditional method of "store and then analyze later" approach. The reason for this is obvious: firstly, such a large amount of data itself is not easy to manage, and secondly, by

Software Architecture for Big Data and the Cloud. DOI: 10.1016/B978-0-12-805467-3.00011-9

the time one will start the analysis, data may lose its value. Since the data almost needs to be processed in a real-time manner, the latency in processing the data should be quite low when compared to batch data processing systems such as Hadoop.

The limitation of the previous approach in managing and analyzing high volume and velocity data in real-time has led to development of sophisticated new distributed techniques and technologies. The processing of data with such features is defined as "*data stream processing*" or "*stream processing*" or sometimes called stream computing. The research in the area of stream processing can be divided into three areas:

- Data stream management systems where online query languages have been explored;
- Online data processing algorithms where the aim is to process the data in single pass; and finally
- The stream processing platforms/engines, which enable implementation and scaling of stream processing-based applications.

Given the high business demand of stream processing platforms, in this chapter, we are specifically focused on the analysis of different stream processing platforms/engines and developing taxonomy of their different features. Current stream processing platforms/systems borrow some features from dataflow systems developed in the 1960s to low level signaling networks developed during the 1980s [38], and then from data stream management systems developed during the 1990s. Stream processing platforms enable specifically the execution and deployment of real-time data processing applications in a scalable and fault-tolerant manner.

In recent years, due to different challenges and requirements posed by different application domains, several open source platforms have emerged for real-time data stream processing. Although different platforms share the concept of handling data as continuous unbounded streams, and process them immediately as the data is collected, they follow different architectural models and offer different capabilities and features. For both research and business communities who want to adopt the stream processing technology, it is important to understand what capabilities and features different stream platforms offer to its end users.

There are several studies that have been conducted on stream processing platforms and large-scale computing platforms. For example, Jagmohan Chauhan et al. [11] have conducted an assessment on Yahoo! S4's technical performance regarding its scalability, loss of events, and fault tolerance. The Samza team performed its own comparison of Samza with other platforms such as Storm and Spark Streaming. Chen et al. [12] have also proposed a brief survey of stream computing platforms such as S4 and Storm. However, currently there is not a single study that synthesizes the features of different stream processing systems and developed a comprehensive set of standard criteria for evaluating and understanding those platforms particularly from point of view of both technical and business issues such as cost. In this chapter, the basic research question that we are answering is "*how to develop a framework that can help the end users to classify different data stream processing platforms using their different characteristics*."

Therefore, to fill this gap, in this chapter, we propose a taxonomy of different features of stream processing platforms that are important from both research and business perspectives. The taxonomy is then used to compare existing stream processing systems to survey current research developments and enable identification of possible future development. As it is not easy to get access to commercial stream processing platforms, the proposed taxonomy is derived after studying different open-source

platforms including Data Stream Management Systems (DSMS), Complex Event Processing (CEP) systems, and stream processing systems such as Storm.

The rest of the chapter is organized as follows. Section 11.2 gives a brief background about Stream processing systems/engines. In Section 11.3, we present the taxonomy and different set of criteria that has been utilized to build the taxonomy. Section 11.4 presents a survey of stream processing platforms and other closely related platform. Section 11.5 presents the comparison between the platforms. Section 11.6 concludes the chapter with future directions and research questions.

## 11.2 STREAM PROCESSING PLATFORMS: A BRIEF BACKGROUND

The stream processing research area has a long history of development that started at least five decades ago. The topics that underpin it include distributed computing, parallel computing, and message-passing. Stephens [38] shows that the earliest research about stream processing can be tracked back to the 1960s in the form of dataflow systems or data management systems. However, during that period only some theoretical development took place without any implementation of a dataflow management system. In the late 1960s, the primary solutions for dataflow management were Database Management Systems (DBMSs) [4], but there were some starting developments to integrate features required for the data streams with the design models of DBMSs. For example, some DBMS at that time implemented a distributed network model in which the different nodes in the network were used to undertake different computational tasks [4]. In the 1970s, researchers proposed new approaches to describe and process dataflow. In 1974, the first dataflow programming language, Lucid, was introduced to allow for limited use of data streams [6]. One important contribution of that decade is the Kahn's Processing Network (KPN) that proposed a distributed computing network model. In the same year, Kahn [18] introduced a network processing model for modeling distributed systems, particularly using First In, First Out (FIFO) communicating channels. The FIFO communication strategy which represents a queuing behavior of input data has been one of the dominant data scheduling strategies over the years. Other than modeling a distributed system, KPNs model was also proven to be useful for modeling signal processing system, which is generally recognized as another early version of stream processing platform [38].

The 1980s was a decade of the booming development of stream processing theory, during which another significant concept, namely synchronous concurrent computation, was introduced [38]. The term synchronous concurrent computation is another way of saying parallel computing. This technology further led to the new advancements in data management technologies to enhance efficiency of different queries, named as reactive systems [38]. The reactive systems used highly concurrent lightweight message passing to quickly respond to the input data events from sensors. The real time behavior of data ingestion components in the reactive systems is one of the most remarkable characteristics of todays' stream processing platforms. In the early 1990s, another development began in the area of DBMS that was started to serve the requirements of applications such as network traffic analysis where data is generated in real time, and continuous queries were required. This development led to introduction of features such as continuous querying, temporal data models which were integrated in the form of Data Stream Management Systems (DSMS). These systems allow persistent queries on both continuous, real-time stream of data and standard database tables. The key feature of these systems is that data can be queried on the fly before it is stored permanently.

The development of these topics progressed during the last five decades, and not until recently that they grew both theoretically and physically mature enough to construct a modern blueprint of Stream processing systems that we are using today. Stream processing platforms are designed to run on top of distributed and parallel computing technologies such as clusters to process real-time stream of data. Dataflow has always been the core element of stream processing system. *Data Streams* or *Streams* in the stream processing context refers to the infinite dataflow within the system. Logically, a stream processing platform is a message-passing system [11], whose data processing activities are driven by the owing data [9] so that the topic of data passing and dataflow management have been drawing all the attention of stream processing researchers over the years.

### 11.2.1 REQUIREMENTS OF STREAM PROCESSING PLATFORMS/ENGINES

Stream processing is also a solution for the management of big data using different approach than that of batch processing. Michael et al. [39] proposed a general set of requirements for data stream processing engines, which are listed as follows:

- (R1) Keep the data moving: Keep latency at absolute minimum by processing data as soon as captured.
- (R2) Handle stream imperfections: Provide built-in features for handling missing or out-of-order data.
- (R3) Query using SQL on Streams: Allow SQL queries on data streams to build extensive operators.
- (R4) Generate predictable outcomes: Be able to provide guaranteed repeatable and predictable results.
- (R5) Integrate stored and streaming data: Be able to combine different data streams from external sources ingested with different speed.
- (R6) Guarantee Data safety and Availability: Provide fault tolerance features to ensure continuous processing of data.
- (R7) Partition and scale applications automatically: Distribute data and process among multiple processors/CPUs/nodes.
- (R8) Process and respond instantaneously: Achieve the real-time response with minimal overhead for high-volume data streams.

Most stream processing systems are built more or less based on these requirements and essentially work with moving data (data streams) and do the processing in the memory. For example, Yahoo! S4 is claimed to have features like real-time response, as well as being distributed, fault-tolerant, scalable, and pluggable [11] that address the requirements R1, R3, R4, R6, R7, and R8. However, there requirements are not necessarily perfectly encompassed by a stream processing platform as a whole since some of them are in conflict with each other. Bockermann [9] states that the activity of guaranteed data safety and availability (R2) will raise the cost of computation and storage performance, which eventually compromises the throughput of dataflow, eventually leading to adverse effects on latency and real-time response (R1, R8). These conflicting requirements result in some differences in the way they have been emphasized in different stream processing platforms. But in general these requirements are commonly addressed in the design of today's stream processing engines. In addition to these requirements, there are other requirements that are essential both from developer and business perspective

before any stream processing platform/engine is adopted for organizational use. Our proposed taxonomy concerns not only the technical design of a stream processing engine, but also tries to point out the criteria essential for commercial adoption. The additional requirements that can be used for the comparison of different stream processing platforms will be discussed in the later sections.

### 11.2.2 GENERIC MODEL OF MODERN STREAM PROCESSING PLATFORMS/ENGINES

The modern stream processing platform/engine is built on top of distributed and parallel computing technologies. The typical distributed and parallel computer architecture, arranged as a cluster, can be found in all modern stream processing systems like Yahoo! S4, Twitter Storm, and LinkedIn Samza [11,29,33].

The cluster computing technology allows a collection of connected computers to work together providing adequate computing power for the processing of large datasets. Such a feature is so important for stream processing platforms that clusters have become one of its necessary modules for processing high velocity big data. A typical stream processing platform ingests data from external sources such as Facebook, Twitter and databases, and delivers processed results either back to storage or to be published on an online system (or website). Yahoo! S4 [11] is good example where a computer cluster is utilized for large real-time dataset processing. In the example, the S4 platform uses an additional adapter to preprocess raw input data into data events for the cluster with the purpose of better scheduling of event processing tasks [11]. The figure presents an external view of a stream processing platform which can be regarded as a general structure model.

A stream processing platform, in its nature, is a data-passing system, thus the execution module of a stream process platform/engine can be modeled using the dataflow from the sources where data is ingested to the sink which will output the processed data. However, the dataflow in stream processing is quite different from the one in batch processing. According to Babcock et al. [7], the stream processing platforms differ in the following ways:

- Source: data arrives from online sources rather than physical storage like disk or memory.
- Order: Data order and event processing order cannot be controlled by the system.
- Size: data streams are potentially infinite in size, being continuous dataflow.
- Retrieval: data elements generally will be discarded after processing, making it hard to retrieve.

Because of these different features, stream processing systems cannot be designed using the conventional relation model for data management, which is more suitable for the management of static and stored content [7]. Thus, stream processing requires a dynamic model for data management and processing since a stream processing system is in its nature a message-passing system and its activities like programming are driven by the data [9]. As a result, a typical stream processing engine consists of three fundamental elements with respects to the dynamic nature of stream processing system: a source which allows external data passing into the system, a processing node where computation activities are performed, and a sink that passes the processed data out of the system. Fig. 11.1 [9] shows the mechanism of how each individual element addresses the issue of passing data, while in Fig. 11.2, a model combining all the elements shows dataflow execution within a typical stream processing platform.

These elements can be found in all the surveyed stream processing platforms, but some may use a different set of terminology to describe such elements. For instance, while Spring XD inherits totally

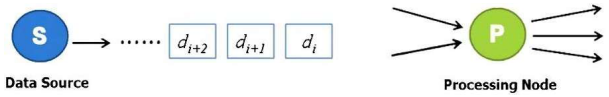


**FIGURE 11.1**

The concept of a data source and a processing node

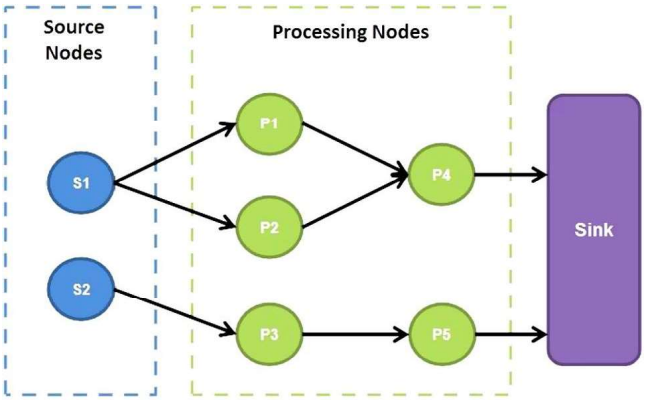


**FIGURE 11.2**

Dataflow model of a stream processing platform

the same nomenclature, Twitter Storm [29,43] uses spouts for naming the sources, and bolts for the processing nodes.

These different elements are connected following a hierarchical model shown in Fig. 11.2, and each of them implements different stream processing components to perform their specific function. For example, a source node implements an event scheduler to allocate input data tasks to appropriate processing nodes in the cluster, while the processing nodes implements applications to consume and process the data. A real world example is Samza, which uses Apache YARN to perform the tasks like data ingestion and scheduling, while in its cluster, machines will use the "Samza API" to process data events [33].

## 11.3 TAXONOMY

In the previous section, we gave a general overview of the requirements and design concepts of modern stream processing platforms. However, as mentioned in the previous section, the existing stream processing platforms/engines differ from each other by emphasizing on different processing requirements. Such differences will make the selection of an appropriate stream processing platform difficult. In this section, a taxonomy framework is proposed to characterize and classify various features of stream processing platforms based on two primary aspects as shown in Fig. 11.3: functional and nonfunctional

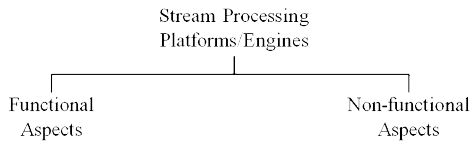


**FIGURE 11.3**

High level classification of stream processing platforms

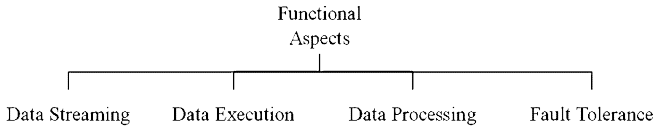


**FIGURE 11.4**

Functional aspects

aspects. The ultimate goal of our research is to provide a solution for better stream processing platform selection, so that taxonomy has been developed considering both functional and nonfunctional aspects of a stream processing platform. The functional criteria are from a stream application perspective and are about the technical aspects of the stream processing platforms, i.e., what features they have and how they perform. The nonfunctional criteria are related to the qualities of the systems and services other than their functional performances.

## 11.3.1 FUNCTIONAL ASPECTS

Christian Bockermann [9] identifies two basic functional components of stream processing platforms as:

1. A queueing or message passing component that is responsible for the communication between processing nodes; and
2. An execution component that provides the runtime and environment for the execution of processing nodes.

In this chapter, an extended taxonomy (Fig. 11.4) of stream processing platforms' technical aspects is proposed, in which Issue 1 has been identified as Data Streaming, and Issue 2 has been divided into two parts as Data Execution and Data Processing. In addition, we identified that the issue of fault tolerance is becoming extremely important to today's stream processing platforms, thus it is regarded as one of their fundamental functionalities, which in Bockermann's survey was studied as one of the challenges in stream processing under the discussion of execution environment.

### *11.3.1.1 Data streaming*

As mentioned earlier, a stream processing system is logically a message-passing system, which means that the communication between different stream processing modules is critical to the entire performance of a stream processing platform. There are two fundamental considerations in relation to

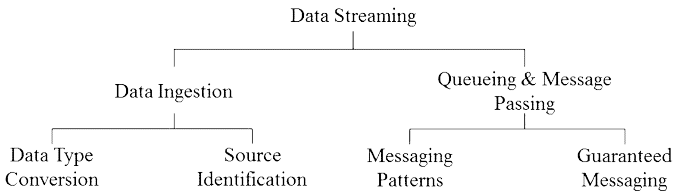


**FIGURE 11.5**

Data streaming criteria

data streaming. One is the ingestion of data, and another is the queuing & message passing (see Fig. 11.5). Data ingestion is about how to collect data from external sources, while queuing and message passing provides the communication channels which connect different nodes within the system.

**Data ingestion.** Data ingestion is related to collecting data from identified sources, but it is different from data collection. The difference lies in the additional activity of formatting the data. The ingestion process often involves the addition and alteration of data content and format with the purpose to use it properly later. The reason for this is that at present, the data that are about to be processed presents a high diversity in their types, while a stream processing platform can process only a few particular types of data. This conflicting situation requires the data being processed satisfy the type requirement of stream processing platforms, so that a process of type identification and conversion is needed by a stream processing platform. Currently, some messaging systems, such as Apache Kafka [27], Apache Flume [22], and ZeroMQ [2], provide this type of conversion service as one of their fundamental components to their users whose time is saved from developing special programs to ingest data from conventional sources such as TCP sockets and HTTP POSTs [37]. Data ingestion process involves another functionality, which is acknowledged as the identification of data sources. In general, there are two types of data sources: stored data and streaming data. Although stream process platforms are primarily designed for the processing of large streaming datasets, sometimes stored data is needed to conduct a comprehensive data analysis. Michael Stonebraker et al. [39] use terms "past" and "present" to describe the features of these two categories of data respectively. They imply that in some cases (e.g., in the analysis of "unusual" events) the integration of both "past" and "present" data is so important to the data analysis that they list it as one of the basic requirements of stream processing platforms (R5). Thus, the data ingestion process of a stream processing system should not only be able to ingest real-time data from online sources, but also to ingest data from physical data storage, such as disks and hard drives.

**Queueing & message passing.** Queueing and message passing is responsible for the communication between different stream processing modules. In Twitter Storm, the streaming modules, spouts and bolts are connected by communication channels provided by ZeroMQ messaging library [26]. In a real-world perspective, different stream processing platforms/engines implement different messaging systems with various patterns of messaging. For distributed architecture, there are four typical message patterns which are widely used [21]:

- *PAIR:* message communication is established strictly within one-to-one peers.

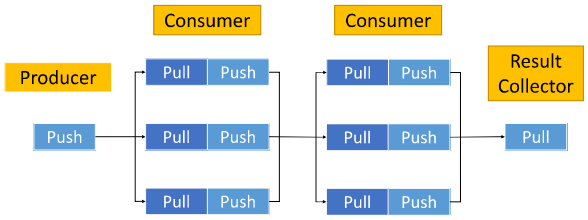


**FIGURE 11.6**

Push/pull messaging pattern

- *Client/Server:* messages are distributed from server according to the requests sent from clients.
- *Push/Pull (pipeline)*: enables the distribution of messages to multiple processors, arranged in pipeline.
- *Publish/Subscribe:* enables the distribution of messages to specific receivers who are interested in the massage content rather than the senders of message.

However, not all of the message patterns are suitable for stream processing which needs to handle high velocity and volume data. For example, the PAIR message pattern's passes messages on one-to-one basis that limits the distribution of large volume of messages. Thus, this pattern is not suitable for a processing node that requests specific data from sources due to unpredictable nature of data content and format. Compared to PAIR and Client/Server, the remaining two patterns do not have the above mentioned problem, making them a common choice of stream processing systems. Push/Pull pattern is a one-way stream processing pattern, in which the downstream nodes will be invoked whenever the upstream nodes finish processing of tasks [28]. As shown in Fig. 11.6, streams go through several processing stages, with no messages' sending upstream.

Pub/Sub is so far the most popular messaging pattern. There is a long list of queueing and messaging systems using pub/sub pattern, such as RabbitMQ [32], Open AMQ [46], Apache ActiveMQ [35], and Apache Kafka. These messaging systems share a common feature that they all have a broker that manages the distribution of messages. In a pub/sub system, streams are called topics. Specifically, a broker is a program that will function like a mediator, which matches the topics subscribed by receivers to the topics published by producers. Fig. 11.7 presents the role of brokers in pub/sub messaging system. Each application could be either the publisher or the subscriber, and the broker will transmit the topics from the publish end to the subscription end.

#### 11.3.1.2 Data execution

A stream processing platform/engine must have an execution component that arranges the data events to appropriate processing nodes. As shown in Fig. 11.8, data execution can be divided into different components such as the scheduling, the scaling, and the distributed computing of data events.

**Scheduling.** Falt and Yaghob [16] observed that the operation order of processing applications is "closely related to the utilization of the hardware components", indicating that the outcome of the task scheduling will consequently influence the overall performance of the stream processing system. In general, most stream processing platforms, thus, have adopted some scheduling solutions. For exam-

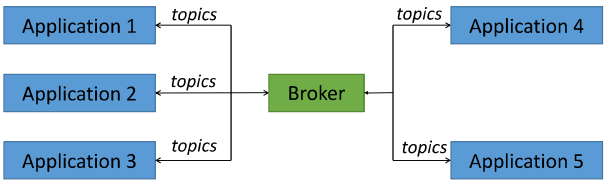


**FIGURE 11.7**

Pub/sub messaging pattern with a broker

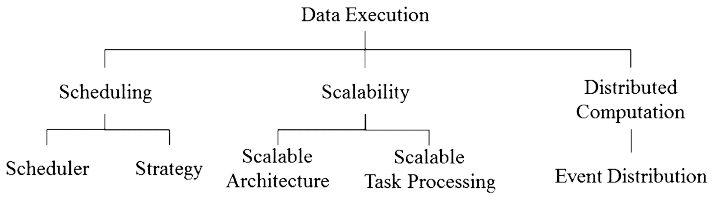


**FIGURE 11.8**

Data execution criteria

ple, Apache Samza is using the Resource Manager provided by YARN, which contains a pure scheduler application that allocates the computation resources [33], while Yahoo! S4 does not use a scheduler application but use Keys to correlate the processing elements to corresponding processing nodes [11]. Other than the scheduling system, the scheduling strategies will also impact the system's performance in terms of queue size, latency, and data throughput. A stream processing system should allow the users to implement the scheduling strategies, or provide options for users to change the strategy implementations. We survey that there are three commonly applied scheduling strategies:

1. *FIFO:* a scheduling strategy which is the acronym of First-In-First-Out, with which the data events are processed in the sequence of their arrival [25].
2. *Capacity:* a strategy that guarantees the minimum capacity of queue processing, while if there are available resources, it will allocate it to the waiting queues to enlarge its capacity.
3. *Fair:* a strategy that allocates almost the equal capacity to each active task slot, making each job get approximately the same amount of processing time and resources [34].

There is no best strategy or best scheduler. Jiang and Chakravarthy [25] state that, although the FIFO strategy enables high throughput and low latency scheduling pattern, it did not consider the optimal utilization of hardware resources (e.g., CPU, memory, etc.). However, they observed that the capacity-based scheduling strategy may suffer from the overflowing buffer of data events if the volume of tasks is high. Thus, instead of predetermining a scheduling strategy, it is a better solution to allow users to make the decision. For example, Apache Samza has provided a pluggable API to enable the users to integrate their custom scheduler applications [33]. Other streaming engines, like Spark Streaming and Spring XD, enable users to develop scheduling strategies based on their own scheduler patterns (Batch Scheduler, and Cron Scheduler) [37,48], and Twitter Storm which used to use Nimbus to schedule tasks [29] has now the feature of a pluggable scheduler.

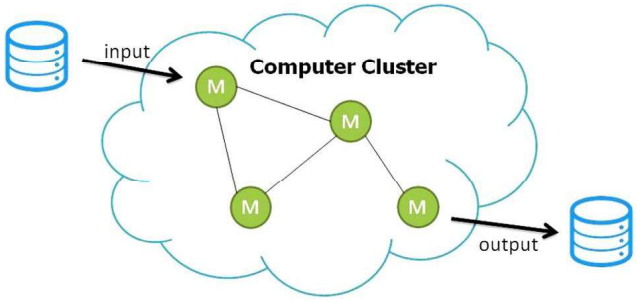


**FIGURE 11.9**

Simple model of a distributed system

**Scalability.** Scalability refers to the ability of a multiprocessor system to handle the increasing amount of tasks, or the ability to enlarge the system itself to tackle the growth of tasks [10]. Scalability is one of the most important requirements in the age of big data because systems need to handle the pressure from incoming high volume data sets. A scalable system should scale both in its architecture and its data processing capacity. A scalable architecture connotes that the system can manage the growth of processing nodes without technical and functional errors, such as data loss and node failures. The important challenge faced by a scalable system is the coordination of the nodes within the expanding network to minimize the loss of data, and the monitoring of the status of each node to avoid the risk of significant node failure [47]. Most of the current stream processing platforms such as Storm, S4, Spark Streaming, and Samza provide scalable systems that allow the enlargement of their architectural topology. The scalable capacity, or scalable task processing, is a result of the enlargement of architecture. With added machines, a cluster should be able to perform more tasks with higher throughput.

**Distributed computation.** Distributed computation is the key component of data execution in stream processing. It is a hard fact that a stream processing system must be a distributed system, which is defined as a collection of independent computers that appears to its users as a single coherent system [42]. Network technologies are applied to realize such coherency by connecting machines within a cluster. In practice, distributed computing in stream processing is responsible for the allocation of jobs and the coordination of the machines in order to perform the event processing in a concurrent manner. It is the core and the foundation of a stream processing platform, the key to realize the processing of lager volume big data. Although different stream processing platforms are different in the design of their cluster topology, the designs are all built up on top of a distributed system pattern shown in Fig. 11.9. Machines in a distributed computer cluster are autonomous and execute the applications whenever events arrive. For example, Yahoo! S4 is using the keyed attributes as the trigger of operations so that a processing node (machine) can autonomously execute the application if an event with corresponding keyed attribute arrives [31].

#### *11.3.1.3 Data processing*

The data processing component of a stream processing platform is a group of programs and applications that is implemented on the computing machines to perform tasks like processing and analysis. The most

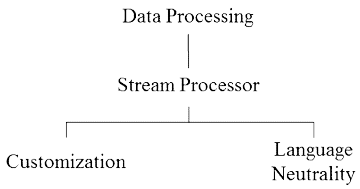


**FIGURE 11.10**

Data processing criteria

important part of data processing in stream processing platforms is the stream processor, which in its nature is a program that can access the streams and invoke appropriate processing applications based on its algorithmic coding, as shown in Fig. 11.10.

**Stream processor.** The term "stream processor" has a different meaning from commonly understood meaning of processors as central processing unit (CPU) or graphics processing unit (GPU). The latter processors are physical, while the stream processor refers to a programing code that provides the communication between data streams and stream processing applications to perform processing. Such a program is also known as application programming interface (API), which is defined as a language and message format used by an application program to communicate with the operating system or some other control program such as a database management system (DBMS) or communications protocol. However, a real time stream processing system does not necessarily include a DBMS or APIs to enable ingestion of the data stream directly. For example, Twitter's streaming API provides connection from user's node to the Twitter's stream of tweet data, whose process is like downloading an infinite file. The stream processor can be described further using two aspects: customization and language neutrality. The stream processing platforms should allow users to develop custom processing applications on their nodes so that they can perform analysis in accordance to their expectations. Currently, all of the surveyed platforms in this chapter provide this feature that allows the users to write streaming applications based on provided patterns. For example, Spark Streaming provides Spark's language-integrated API to users which supports common programming language Java and Scala [36].

The application development using APIs should be compatible with different languages to allow users with different ability to harness the power of stream processing platforms. We choose to call this compatibility as "language neutrality" referring to how many different programming languages a stream processing platform accepts. Java is currently being the most popular programming language which is supported by all the surveyed stream processing platforms, primarily because of its features being simple, stable, and sustainable. Other popular languages used for steaming API and application development include Scala, Python, and so on, among which JVM-based languages take up a significant proportion.

#### *11.3.1.4 Fault tolerance*

The fault tolerance is identified as one of the most important requirements and a challenge for stream processing systems. Fault tolerance is the concept that is derived from the design of a distributed system, which considers that the failure of a single or multiple processing nodes should not impact the correct operation of the entire system. Conventionally, fault tolerance is described as a system's ability

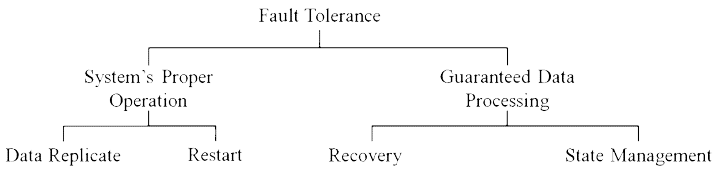


**FIGURE 11.11**

Fault tolerance

to replicate important data before a failure happens, and to restart the failed node to make it functional again. However, currently fault tolerance also includes a new concept of guaranteed data processing, which ensures the data will be fully processed even though a failure happens. Briefly speaking, fault tolerance concerns two things: the proper operation of the entire system and the guaranteed processing of data messages, as shown in Fig. 11.11. The fault tolerance design of a distributed system has to handle different failure situations to maintain proper operation of the system. In a distributed architecture, there will be two situations of failure: the failure of centralized components within the cluster, which are commonly known as the processing nodes, and the failure of parallel applications that perform data processing, which run on the worker nodes within the cluster [29,44]. While a node within the cluster fails, the primary solution is that the system should be able to replicate the key data and reassign the data processing tasks to another node. However, if an application fails to respond, the system should be able to restart the node that runs the application, in order to recover the application from complete failure.

Besides the correct operation, fault tolerance in a stream processing system also guarantees the completeness of message processing, which means that it tries to prevent the system from losing data. Guaranteed data processing is actually a guarantee of the message delivery in each surveyed platform. For example, Twitter Storm uses the combination of Kestrel queue and time-out function to manage the state of message, in which the message will not enter the processing stage until the previous message has been fully processed, or fail to process completely within given timeout [29]. Other platforms may use a messaging system such as Kafka [27] to guarantee that the message is delivered at least once. The "at least once" messaging approach ensures that there is no data loss, but it is likely to cause duplication in some fault scenario, in which the same message could be sent twice to the consumer [33].

### 11.3.2 NONFUNCTIONAL ASPECTS

While the functional requirements concern the technical behavior of a stream processing system, nonfunctional requirements concern the results and effects of its behavior. Nonfunctional requirements define the overall qualities or attributes of the resulting system, and they include safety, security, reliability, and some management issues such as costs, time, and schedule. The most important nonfunctional criteria for selection of a stream processing system are cost, technical support, and the user community (see Fig. 11.12).

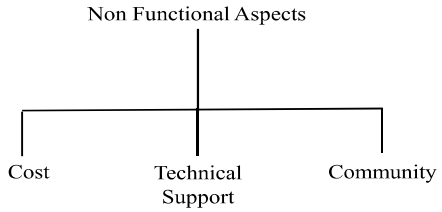


**FIGURE 11.12**

Nonfunctional aspects

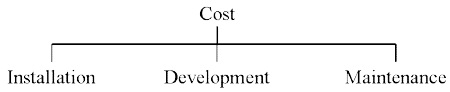


**FIGURE 11.13**

Stream processing platforms' cost

#### 11.3.2.1 Cost

The cost of a stream processing platform consists of several components; each refers to different usage stage. As shown in Fig. 11.13, installation, development, and maintenance are the three basic stages that a system has to go through, and each of them involves a significant amount of cost. Installation cost includes the licensing cost and installation support cost. However, as the target systems in our survey are all open-source, there is no licensing cost for each of them, but still it is important to include such cost into the study because it will help in development of a decision making framework that can be generalized to suit the selection of all kinds of stream processing platforms. Also it may include the cost of hiring a specialized professional for installation support. For example, the average on-going development and maintenance cost of Apache Storm [26] counts to about US $1,700 million per three years [24], which is so significant figure that a consumer cannot neglect. Among the cost, payments to experts and experienced personnel take up the largest proportion.

There will be development and maintenance costs after the installation of a stream processing platform. Users have to develop applications on top of the platforms to fit their particular purpose for big data analytics. The primary activities involved in the development stage include planning and programming, which requires the involvement of professional personnel whose payments take up the largest proportion of development cost. Besides, the stream processing platform is a complex system consisting of numerous hardware and software, which requires careful and on-going maintenance to ensure its proper operation. The implementation of a stream processing platform is always an important decision from a long-term perspective, so that users have to carefully address the problem before selecting one solution.

#### 11.3.2.2 Technical support

Technical support is the service that the stream processing platform developer can offer to users. Similarly to how we classify the cost, a classification of technical support is based on the stages involved, as shown in Fig. 11.14. From the users' perspective, the essence of technical support is the availability

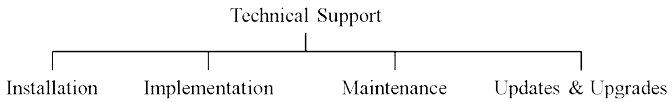


**FIGURE 11.14**

Technical support criteria

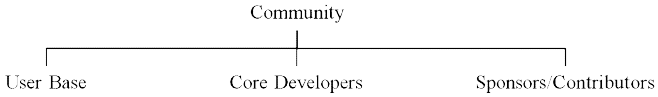


**FIGURE 11.15**

Community support criteria

of instruction and consultation that can be offered by the platforms' technical team or the community. Besides, how convenient it is for the consumers to reach the support is another consideration. In short, technical support is one of the most important factors that contribute to the overall customer satisfaction, which from a commercial perspective indicates whether a platform is successful or not.

#### *11.3.2.3 User community*

One also needs to study the community support available for the stream processing platforms. There are three primary communities included in the community aspect: the user community, the developer community, and the contributor community, as shown in Fig. 11.15. These communities can provide essential support in regards to the issues such as development, operation, and maintenance of the stream processing platforms. The user base of a platform reflects its popularity. The difference of popularity between each platform is very likely to be a result of the comparison of their functionalities. A larger user base always means that there will be more feedback about a particular platform that a user can access and utilize. Feedback is extremely meaningful when users attempt to gain deep insight into the platform's performance from either the advantageous or the disadvantageous side. The core developers and contributors of a stream processing platform will be the source of supporting information. Users can acquire technical information like manual, update, and upgrades from these sources, and sometimes users can ask for a change or update in case of issues such as bugs or their individual use case.

## 11.4 A SURVEY OF STREAM PROCESSING PLATFORMS

In the previous section, we presented a taxonomy framework that can be used to judge the functional and nonfunctional features of a stream processing platform. As we always stressed, the goal of our research is to enable future development in this research area and also to enable selection of proper stream processing system based on user's need. In this section, we conduct a survey of different stream processing platforms based on our proposed taxonomy. The survey includes three versions of stream processing systems: Data Stream Management System (DSMS), Complex Event Processing (CEP) system, and stream processing platform/engine. All of them can be identified as a stream processing

system, but they differ in their goals. DSMS aims to handle the continuous online queries with predefined query language like CQL [17], which are more closely related to that of a DBMS. Complex event processing systems have a higher goal to identify the important events through stream data analytics. Both DSMS and CEP systems tackle a limited range of data types [5,30]. The stream processing platforms in general can handle a broader range of different formats.

### 11.4.1 DATA STREAM MANAGEMENT SYSTEMS

The Data Stream Management Systems (DSMS) are the earliest version of the stream processing systems. They encompass the functions of a conventional database management system (DBMS), but further offer the ability to execute continuously incoming data queries to enable real-time data processing. The evolution of DSMS from DBMS is a result of the world's demand for high velocity data querying, especially in the industries where sensors are heavily used [17]. DSMS is not the key focus of this article; one can find more details about DSMS from the following references. Arasu et al. [5] give a detailed introduction about the models and issues related to Data Stream Management Systems which is a valuable reference of the important concepts related to DSMSs. Geisler further presents the general architecture of DSMS [17]. There are several other data stream management systems that are developed for different purposes. For instance, there is a distributed spatiotemporal DSMS named PLACE which is used for monitoring moving objects, and XStream is used for the tracking and processing of signals [45,19]. Other systems, like Borealis, NILE and Medusa, are designed as general purpose data query processing DSMSs [20,1].

### 11.4.2 COMPLEX EVENT PROCESSING SYSTEMS

The Data Stream Management System provides a solution for real-time data query processing. However, as DSMS handles only generic data without identifying interesting situations known as events. Thus, the Complex Event Processing (CEP) systems have been developed for this purpose. In [14], the improvements of CEP upon DSMS have been explained in detail; however, the most essential development in CEP systems is that they associate semantics with the data so that the system can detect, filter, and combine the data, or so-called event notifications from external sources to understand the events. Compared to generic queries answered by DSMS, events are higher-level analytics results which enable people to better understand their environments and take appropriate actions.

Though CEP is not capable of processing of unstructured data compared to stream computing platforms, it is efficient in analyzing the structured or semistructured data. Currently there are many CEP systems that are available for either business or research purpose, e.g., Oracle Event Processing, Microsoft StreamInsight [3], SQLStream s-Server [23] and StreamBase [40]. Besides the systems introduced above, there are other CEP platforms that share a similar set of characteristics, such as Esper, Cayuga, and Apama [15,41]. These systems have a bit different architecture than those surveyed, but overall they are all following the same concept of a stream processing system.

### 11.4.3 STREAM PROCESSING PLATFORMS/ENGINES

#### 11.4.3.1 Apache storm

Storm is a distributed stream processing platform that was originally created by Nathan Marz and then acquired by Twitter who turned Storm into a free and open source streaming engine. Currently the Storm project is undergoing incubation in the Apache Software Foundation, sponsored by the Apache Incubator. Storm's architecture model is commonly known as topology, which mostly follows the general pattern we mentioned in the previous sections. The source node in Storm is named spout, which emits data into the cluster. Processing nodes in Storm are called bolts. The data stream in storm is an unbounded sequence of tuples having a formatted structure. Storm topology is a parallel architecture in which nodes execute concurrently to perform different tasks. Users can specify the degree of parallelism for each node to optimize the resources spent on tasks with different complexity.

#### 11.4.3.2 Yahoo! S4

S4 (Simple Scalable Stream Processing System) is a distributed real-time data processing system developed by Yahoo. Yahoo! S4 architecture is inspired by the MapReduce model. However, unlike MapReduce which has a limitation on scaling, Yahoo! S4 is capable of scaling to a large cluster size to handle frequent real-time data [11]. Similar to other distributed and parallel systems, Yahoo! S4 has a cluster consisting of computing machines, known as processing nodes (PNs). A processing node is the host of processing elements (PEs) which perform data processing tasks on events. The data stream within S4 is a sequence of events. The adapter is responsible for the conversion of raw data into events before delivering the events into the S4 cluster. When a processing node receives input events, it will assign it to associate PE via the communication layer. ZooKeeper plays a role as PN coordinator to assign and distribute events to different PEs in different stages.

#### 11.4.3.3 Apache Samza

Samza is the stream processing framework of LinkedIn, and it is now another incubator project of Apache Software Foundation. The platform is scalable, distributed, pluggable, and fault tolerant, and it has recently released its new version as an open source project. However, the new version has some limitations on its fault tolerance semantics according to the Samza development team. The framework is highly integrated with Hadoop YARN, the next generation cluster manager, so that the architecture of Samza is very similar to that of Hadoop. Samza consists of three layers: a streaming layer, an execution layer, and a processing layer [33]. Samza heavily relies on dynamic message passing and thus includes a message processing system like Kafka, adopted for data streaming.

#### 11.4.3.4 Spring XD

Spring XD is a unified big data processing engine, which means it can be used either for batch data processing or real-time streaming data processing. It is now licensed by Apache as one of the free and open source big data processing systems. The goal of Spring XD is to simplify the development of big data applications. The Spring XD uses cluster technology to build up its core architecture. The entire structure is similar to the general model discussed in the previous section, consisting of a source, a cluster of processing nodes, and a sink. However, the Spring XD is using another term called XD nodes to represent both the source nodes and processing nodes. The XD nodes could be either the entering point (source) or the exiting point (sink) of streams. The XD admin plays a role of a centralized tasks controller who undertakes tasks such as scheduling, deploying, and distributing messages. Since

Spring XD is a unified system, it has some special components to address the different requirements of batch processing and real-time stream processing of incoming data streams, which refer to taps and jobs. Taps provide a noninvasive way to consume stream data to perform real-time analytics. The term noninvasive means that taps will not affect the content of original streams.

#### *11.4.3.5 Spark streaming*

Spark Streaming is an extended tool of the core Spark engine to enable this large-scale data processing engine to process live data streams. The role of Spark Streaming is very similar to the client adapter used by Yahoo! S4. Currently, Spark Streaming is developed by Apache Software Foundation, which is responsible for the testing, updates, and release of each Spark version. It is important to know that Spark Streaming is not the data processing engine. The engine is named Apache Spark, but without Spark Streaming the engine cannot perform real-time data analytics. Spark Streaming runs as a filter of streams that divides the input streams into batches of data, and dispatches them into the Spark engine for further processing. The Spark engine consists of many machines to form a computing cluster, which is used in a similar manner as in other surveyed platforms. The most special feature of Spark Streaming is how it treats data streams. The data streams within Spark Streaming are called discretized streams, or DStream, which is, in fact, a continuous sequence of immutable datasets, known as Resilient Distributed Datasets (RDDs). All RDDs in a DStream contain data within certain interval, so that they can be clearly separated and ordered for parallel processing.

## 11.5 COMPARISON STUDY OF THE STREAM PROCESSING PLATFORMS

In this section, we combined all the information we collected to perform a comparison study of all the surveyed stream processing platforms. The criteria used for the comparison are based on our proposed taxonomy framework. Tables 11.1–11.4 give an overview of the features of all surveyed platforms. Due to unavailability of public data for nonfunctional aspects such as cost, these tables only reveal the differences for all the functional criteria. In the following section, we compare different stream processing systems emphasizing metrics such as Scalability, Messaging and Distribution, and Fault Tolerance.

### 11.5.1 SCALABILITY

The scalability of a stream processing platform is represented by its ability to integrate and coordinate new processing nodes, as well as the ability to partition the tasks to newly added machines. As shown in Table 11.1, although all the streaming engines claim to be scalable, the ways they scale are slightly different. The first difference is the complexity of scalability. Storm, Samza, Spring XD, and Spark Streaming all allow users to add nodes dynamically, and the engine will automatically define the parallelism for the new nodes. In contrast, Yahoo! S4 requires a redefinition of the nodes' parallelism before they can be added into the cluster, which adds up to the difficulty of scaling.

**Table 11.1 Comparison based on data execution**

| System | | Features | | | |
|---|---|---|---|---|---|
| | | Scheduling | Data execution Scalability | | Distributed computing |
| | | Scheduler | Scalable architecture | Scalable task processing | Distributed event processing |
| DSMS | STREAM | Global scheduler with round-robin scheme | X | X | X |
| | Aurora | A state-based scheduler & a feedback-based scheduler | X | X | X |
| Complex event processing system | SQLStream | A user-defined scheduler | ✓ | ✓ | ✓ |
| | Oracle event processing | Oracle enterprise scheduler | ✓ | ✓ | ✓ |
| | Microsoft StreamInsight | StreamInsight scheduler | ✓ | ✓ | ✓ |
| | StreamBase CEP | – | ✓ | ✓ | ✓ |
| Stream processing platform | Apache storm | Nimbus | ✓ | ✓ | ✓ |
| | Yahoo! S4 | – | ✓ | ✓ | ✓ |
| | Apache Samza | YARN | ✓ | ✓ | ✓ |
| | Spring XD | Cron scheduler | ✓ | ✓ | ✓ |
| | Spark Streaming | Batch scheduler | ✓ | ✓ | ✓ |

## 11.5.2 MESSAGING & DISTRIBUTION

As shown in Table 11.2, the surveyed platforms differ in their concept for messaging, though the publish/subscribe approach is the most popular and adopted by three of them (Samza, Spring XD, and Spark Streaming). The advantage of pub/sub pattern messaging is that it can handle the complexity of input messages from different sources. Differently, Storm and S4 use pipeline pattern for message passing, while Storm is a pull model and S4 is a push model. Both models ensure a fast and direct passing of messages, with low latency. However, the pipeline being a form of linear transmission method makes it unsuitable for distribution of a highly complex volume of messages, compared to pub/sub pattern which is using a broker to coordinate the events according to the demands of users.

## 11.5.3 DATA PROCESSING/STREAM PROCESSORS

As discussed before, most stream processing platforms use JVM which makes Java the main language for programming new applications. However, Storm allows programming in more or less all the languages. SpringXD only allows programming of processing units using Java.

**Table 11.2 Comparison based on data streaming features**

| Systems | | Features | | |
|---|---|---|---|---|
| | | Data streaming | | |
| | | Data ingestion | | Messaging & queueing |
| | | Data type conversion | Source | Messaging queueing pattern |
| DSMS | STREAM | X | Stored data | push model |
| | Aurora | X | Streaming data, stored data | Push/pull model |
| Complex event processing system | SQLStream | ✓ | Streaming data, stored data | Kafka pub/sub model with broker |
| | Oracle event processing | ✓ | Streaming data, stored data | Point-to-point, pub/sub model |
| | Microsoft StreamInsight | ✓ | Streaming data, stored data | Push/pull model |
| | StreamBase CEP | ✓ | Streaming data, stored data | RabbitMQ pub/sub model with broker |
| Stream processing platform | Apache storm | ✓ | Streaming data, stored data | Pull model pipeline |
| | Yahoo! S4 | ✓ | Streaming data, stored data | Push model pipeline |
| | Apache Samza | ✓ | Streaming data, stored data | Kafka pub/sub model with brokers |
| | Spring XD | ✓ | Streaming data, stored data | Kafka pub/sub model with brokers |
| | Spark streaming | | Streaming data, stored data | Kafka pub/sub model with brokers |

### 11.5.4 FAULT TOLERANCE

Fault tolerance is a necessary function of any stream platform to provide reliable processing of big data. As shown in Table 11.4, each surveyed platform in this article ensures the proper operation of the entire system whenever there is a failure of nodes. However, the platforms differ in the way they tackle the issue of data loss. All of the platforms ensure the data can be recovered from faults, but they are using different approaches to realize this goal. While Storm is using a stateless method to reprocess the duplicated data, the other four engines' approach is stateful – recovering the data from the state where the last checkpoint is tracked. In particular, Spark Streaming uses a set of high frequency checkpoints to track the state of data processing, which consequently allows a faster recovery of data as compared to the other three engines which work with checkpoints.

## 11.6 CONCLUSIONS AND FUTURE DIRECTIONS

Big data problems have brought many changes in the way data is processed and managed over time. Today, data is not just posing challenge in terms of volume but also in terms of its high speed generation. The data quality and validity varies from source to source, and thus are difficult to process. This issue has led to the development of several stream processing engines/platforms by different companies such as Yahoo, LinkedIn, etc. Besides better performance in terms of latency, stream processing overcomes another shortcoming of batch data processing systems, i.e., scaling with high "velocity"

**Table 11.3 Comparison based on data processing**

| Systems | | Features<br>Data processing<br>Stream processors<br>Customized operators | Language neutrality |
|---|---|---|---|
| DSMS | STREAM | – | CQL |
| | Aurora | – | – |
| Complex event processing system | SQLStream | – | SQL |
| | Oracle event processing | ✓ | Java. SQL |
| | Microsoft StreamInsight | ✓ | C# |
| | StreamBase CEP | ✓ | Java |
| Stream processing platform | Apache storm | ✓ | Any Programing Languages (JVM or Non-JVM) |
| | Yahoo! S4 | ✓ | Java, C + +, Python, etc. |
| | Apache Samza | ✓ | Only JVM Languages |
| | Spring XD | ✓ | Java |
| | Spark streaming | ✓ | Java, Scala, Python |

data. Availability of several platforms also resulted in another challenge for user organizations in terms of selecting the most appropriate stream processing platform for their needs. In this chapter, we proposed a taxonomy that facilitated the comparison of different features offered by the stream processing platforms. Based on this taxonomy we presented a survey and comparison study of five open source stream processing engines. Our comparison study provides an insight of how to select the best platform for a given use case.

From the comparison of different open source stream processing platforms based on our proposed taxonomy, we observed that each platform offers very specific special feature that makes its architecture unique. However, some features make a stream processing platform more applicable for different scenarios. For example, if the organization's data volume changes dynamically, it is better to choose a platform such as Storm which allows dynamic addition of nodes rather than Yahoo! S4. Similarly, if an organization wants to process all the data that is ingested into the system, the guaranteed data processing feature is what it should look for. In contrast to commercial offerings, organizations can save on licensing fees by using open-source platforms. However, the support given for maintenance of such platforms becomes an important factor in making decisions about adopting a particular platform. The user base and support given for each platform varies quite drastically. Storm has the largest user base and also supports services. Yahoo! S4 comes with almost no support.

Based on the survey, it also becomes clear that the performance of a stream processing system depends on multiple factors. However, the performance will always be limited by the capacity of the underlying cluster environment in which real processing is done. More or less every system that was surveyed does not allow using cloud computing resources which can scale up and down according to the volume and velocity of data that needs to be processed. Moreover, the job scheduling mechanisms used by the systems are not very sophisticated and do not take into the consideration the performance

**Table 11.4 Comparison based on fault tolerance features**

| Systems | | Features | | | |
|---|---|---|---|---|---|
| | | Fault tolerance | | | |
| | | Proper operation of the system | | Guaranteed data processing | |
| | | Replication | Restart | Data recovery | State management |
| DSMS | STREAM | – | – | – | – |
| | Aurora | – | – | – | – |
| Complex event processing system | SQLStream | ✓ | ✓ | ✓ | Stateful management using checkpoints |
| | Oracle event processing | ✓ | ✓ | ✓ | Stateful management |
| | Microsoft StreamInsight | ✓ | ✓ | ✓ | Stateful management using checkpoints |
| | StreamBase CEP | ✓ | ✓ | ✓ | Stateful management using synchronization |
| Stream processing platform | Apache storm | ✓ | ✓ | ✓ | Stateless management |
| | Yahoo! S4 | ✓ | ✓ | ✓ | Stateful management using checkpoints |

of underlying infrastructure which can be quite heterogeneous in some cases. In the future, we would like to conduct a cost and risk analysis of different streaming platforms and conduct a more extensive comparison study. The current taxonomy is derived after studying different open-source stream processing platforms, which limits the scope of our taxonomy. To overcome this limitation, we would also study some key commercially available stream processing platforms such as IBM Stream.

## REFERENCES

[1] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, et al., The design of the Borealis stream processing engine, Cidr 5 (2005) 277–289.
[2] F. Akgul, Zeromq, Packt Publishing Ltd., 2013.
[3] M. Ali, An introduction to Microsoft SQL server StreamInsight, in: Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Application, 2010, p. 66.
[4] H.C. Andrade, B. Gedik, D.S. Turaga, Fundamentals of Stream Processing: Application Design, Systems, and Analytics, Cambridge University Press, 2014.
[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, J. Widom, STREAM: the stanford data stream management system, in: M. Garofalakis, J. Gehrke, R. Rastogi (Eds.), Data Stream Management: Processing High-Speed Data Streams, Springer, Berlin, Heidelberg, ISBN 978-3-540-28608-0, 2016, pp. 317–336.
[6] E.A. Ashcroft, W.W. Wadge, Lucid, the dataflow programming language, APIC Stud. Data Process. 22 (1985).
[7] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2002.
[8] M. Beyer, Gartner says solving 'big data' challenge involves more than just managing volumes of data. URL http://www.gartner.com/newsroom/id/1731916, 2011.
[9] C. Bockermann, A Survey of the Stream Processing Landscape, Tech. Rep. No. 6. TU Dortmund, Germany, 2014, 5.
[10] A.B. Bondi, Characteristics of scalability and their impact on performance, in: Proceedings of the 2nd International Workshop on Software and Performance, 2000, pp. 195–203.

[11] J. Chauhan, S.A. Chowdhury, D. Makaroff, Performance evaluation of Yahoo! S4: a first look, in: 2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2012.
[12] C.P. Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: a survey on big data, Inf. Sci. 275 (2014) 314–347.
[13] M. Chen, S. Mao, Y. Liu, Big data: a survey, Mob. Netw. Appl. 19 (2) (2014) 171–209.
[14] G. Cugola, A. Margara, Processing flows of information: from data stream to complex event processing, ACM Comput. Surv. 44 (3) (2012) 15.
[15] A.J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, W.M. White, et al., Cayuga: a general purpose event monitoring system, Cidr 7 (2007) 412–422.
[16] Z. Falt, J. Yaghob, Task scheduling in data stream processing, in: Dateso, 2011, pp. 85–96.
[17] S. Geisler, Data stream management systems, in: Data Exchange, Information, and Streams, 2013, pp. 275–304.
[18] K. Gilles, The semantics of a simple language for parallel programming, in: Proceedings of the IFIP Congress, in: Information Processing, vol. 74, 1974, pp. 471–475.
[19] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, S. Madden, Xstream: a signal-oriented data stream management system, in: IEEE 24th International Conference on Data Engineering, 2008, ICDE 2008, 2008, pp. 1180–1189.
[20] M.A. Hammad, M.F. Mokbel, M.H. Ali, W.G. Aref, A.C. Catlin, A.K. Elmagarmid, et al., Nile: a query processing engine for data streams, in: Proceedings of 20th International Conference on Data Engineering, 2004, 2004, p. 851.
[21] P. Hintjens, Zeromq: Messaging for Many Applications, O'Reilly Media, Inc., 2013.
[22] S. Hoffman, Apache UME: Distributed Log Collection for Hadoop, Packt Publishing Ltd., 2015.
[23] J. Hyde, Data in flight, Queue 7 (11) (2009) 20.
[24] ITG, Business Care for Enterprise Big Data Deployments: Comparing Costs, Benefits, and Risks for Use of IBM InfoSphere Streams and Open Source Storm, Tech. Rep., International Technology Group (ITG), Santa Cruz, California, 2013.
[25] Q. Jiang, S. Chakravarthy, Scheduling strategies for a data stream management system, in: Computer Science & Engineering, BNCOD, 2004, pp. 16–30.
[26] M.T. Jones, Process Real-Time Big Data with Twitter Storm, IBM Technical Library, 2013.
[27] J. Kreps, N. Narkhede, J. Rao, Kafka: a distributed messaging system for log processing, in: Proceedings of the NetDB, Athen Greece, 2011.
[28] M. Kay, You pull, I'll push: on the polarity of pipelines, in: Proc. Balisage: The Markup Conference, in: Balisage Series on Markup Technologies, vol. 3, 2009.
[29] N. Marz, Storm, Distributed and Fault-Tolerant Real-Time Computation, Tech. Rep. 2014, twitter.com: Twitter.
[30] B. Mozafari, K. Zeng, L. D'antoni, C. Zaniolo, High-performance complex event processing over hierarchical data, ACM Trans. Database Syst. 38 (4) (2013) 21.
[31] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, S4: distributed stream computing platform, in: 2010 IEEE International Conference on Data Mining Workshops (ICDMW), 2010, pp. 170–177.
[32] A. Richardson, et al., Introduction to RabbitMQ. Google UK, Sept. 25, 2008.
[33] A. Samza, Samza documentation, http://samza.incubator.apache.org/.
[34] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The Hadoop distributed file system, in: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010, pp. 1–10.
[35] B. Snyder, D. Bosanac, R. Davies, Introduction to Apache ActiveMQ. ActiveMQ in Action, 6–16.
[36] A. Spark, Spark streaming, https://spark.apache.org/streaming/.
[37] Spring. Spring XD guide, http://docs.spring.io/spring-xd/docs/1.0.0.M4/reference/html/.
[38] R. Stephens, A survey of stream processing, Acta Inform. 34 (7) (1997) 491–541.
[39] M. Stonebraker, U. Cetintemel, S. Zdonik, The 8 requirements of real-time stream processing, SIGMOD Rec. 34 (4) (2005) 42–47.
[40] TIBCO STREAMBASE: A real-time, low latency data processing with a stream processing engine, http://www.tibco.com/resources/demand-webinar/introduction-tibco-streambase-complex-event-processing (last accessed 05/05/2017).
[41] M. Strohbach, H. Ziekow, V. Gazis, N. Akiva, Towards a big data analytics framework for IoT and Smart City applications, in: Modeling and Processing for Next-Generation Big-Data Technologies, Springer, 2015, pp. 257–282.
[42] A.S. Tanenbaum, M. Van Steen, Distributed Systems, Prentice-Hall, 2007.
[43] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, et al., Storm@ twitter, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, 2014, pp. 147–156.
[44] M. Treaster, A survey of fault-tolerance and fault-recovery techniques in parallel systems, arXiv preprint cs/0501002, 2005.