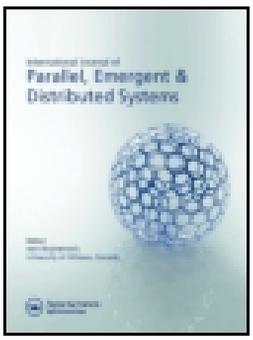

International Journal of Parallel, Emergent and Distributed Systems

ISSN: 1744-5760 (Print) 1744-5779 (Online) Journal homepage: https://www.tandfonline.com/loi/gpaa20

A review on big data real-time stream processing and its scheduling techniques

Nicoleta Tantalaki, Stavros Souravlas & Manos Roumeliotis

To cite this article: Nicoleta Tantalaki, Stavros Souravlas & Manos Roumeliotis (2019): A review on big data real-time stream processing and its scheduling techniques, International Journal of Parallel, Emergent and Distributed Systems, DOI: 10.1080/17445760.2019.1585848

To link to this article: https://doi.org/10.1080/17445760.2019.1585848

INTERNATIONAL JOURNAL OF PARALLEL, EMERGENT AND DISTRIBUTED SYSTEMS
https://doi.org/10.1080/17445760.2019.1585848

A review on big data real-time stream processing and its scheduling techniques

Nicoleta Tantalaki, Stavros Souravlas and Manos Roumeliotis

Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece

ABSTRACT

Over the last decade, several interconnected disruptions have happened in the large scale distributed and parallel computing landscape. The volume of data currently produced by various activities of the society has never been so big and is generated at an increasing speed. Data that is received in real-time can become way too valuable at the time it arrives and supports valuable decision making. Systems for managing data streams is not a recently developed concept but its becoming more important due to the multiplication of data stream sources in the context of IoT. This paper refers to the unique processing challenges posed by the nature of streams, and the related mechanisms used to face them in the big data era. Several cloud systems emerged to enable distributed processing of streams of big data. Distributed stream management systems (DSMS) along with their strengths and limitations are presented and compared. Computations in these systems demand elaborate orchestration over a collection of machines. Consequently, a classification and literature review on these systems' scheduling techniques and their enhancements is also provided.

ARTICLE HISTORY
Received 31 May 2018
Accepted 19 February 2019

KEYWORDS
Big data; stream processing; real-time processing; task scheduling; resource allocation

1. Introduction

Over the past 20 years data has increased in a large scale and in various fields. Managing the produced data and gaining insights from it, is a challenge and possibly a key to competitive advantage, so industries have to find ways to collect and integrate massive data from widely distributed data sources. The rapid growth of cloud computing and Internet of Things (IoT) promote the sharp growth of data even more. Objects contain embedded technology to interact with the external environment and support decision-making. Sensors all over the world are collecting and transmitting data to be processed and stored in the cloud [1].

The term of big data is utilised exactly to refer to this increase in the volume of data that is difficult to store, process and analyse through traditional database technologies. Big data is characterised by the 4 Vs [1,2], namely, volume, variety, velocity and veracity that clearly depict the need for technologies, which require new forms of integration to uncover fast, large hidden values from large datasets that are diverse, complex and of massive scale. Cloud-based data analytics require high-level, easy-to-use design tools for programming applications dealing with huge, distributed data sources [3].

Recently, multiplication of data stream sources (sensor networks, connected devices etc) is observed in the context of IoT. As the amount of such data grows, the need to use models to process and analyse it as it arrives, becomes imperative. Several technologies have emerged specifically to address the challenges of processing high-volume, real-time data. Although systems for managing

CONTACT Nicoleta Tantalaki nicoleta@uom.gr Department of Applied Informatics, University of Macedonia, Egnatias 156, 54636, Thessaloniki, Greece

data streams is not a recently developed concept, it is becoming more important in the mentioned context of IoT. New frameworks have emerged but their differences and ideal use cases are not clear enough. This raised the motivation to answer the following questions:

- How are stream processing challenges faced by technologies in the big data era?
- Which are the main candidate solutions for data stream processing?
- How do they differ in the way they handle streams of data?
- What issues should be taken into consideration when choosing a stream processing solution?

From Stonebraker's et al. [4] analysis and according to related works (for instance [5,6,7,8,9]), the matters of performance and fault tolerance of these systems rise to the top. Table 2 in Section 5.4 presents a full list of such related works. The unpredictable data characteristics and arrival patterns in streams of data pose unique challenges. Retaining state is also critical in most applications involving processing of streams of data. Frameworks should take advantage of the inherent characteristics of parallel and distributed computing to meet these challenges. The demand of elaborate orchestration over a collection of machines becomes imperative but the available frameworks' in-built scheduling techniques are far from optimal. The dynamic nature of cloud environments makes the efficient placement of stream processing tasks on available resources and application adaptation challenging. The following additional queries were raised:

- Which scheduling approaches have been developed in big data stream processing systems?
- Which are the common themes in the available approaches?
- Which parameters affect the scheduling decisions in streaming big data clusters?

This work aims to address the issues raised by the aforementioned questions. In particular, this survey:

- highlights the challenges associated with processing streams of big data;
- provides an overview of the mechanisms that are used to face them;
- compares dominant frameworks based on a list of criteria that arose from the study of related reviews of the state of the art on stream processing [4,10,11,12,13,14,15,16], and based on (1)–(3): *Informs potential users about the criteria to consider when choosing a framework for a specific task*;
- provides an up-to-date overview and classification of task allocation policies that are either incorporated in the processing systems reviewed or are proposed in the literature;
- detects the factors that affect scheduling decisions, and based on (4)–(5): *Supports the design and implementation of effective stream processing frameworks in the future*.

The rest of this paper is organised as follows. Related work is presented in Section 2. Section 3 describes the steps we followed to conduct our research and the reasons for selecting the specific candidates for evaluation. Section 4 refers to stream processing of big data. The available execution models are presented, a number of issues and requirements are discussed and mechanisms used to face them in the big data era are referred. The dominant frameworks are evaluated based on several functionality characteristics and are compared to finally analyse the reasons for selecting each candidate solution. In Section 5, we focus on the matter of task and resource scheduling in stream processing and present a taxonomy of scheduling approaches found in the literature. A discussion on the findings of our survey is also included. Section 6 provides conclusions and future directions.

2. Related work

Any kind of processing and analysis on voluminous data with no specific structure that are produced and arrive to be processed at high-speed demands frameworks with scaling capabilities, able to take

advantage of parallel and distributed computing. Choosing the right platform based on different use cases and needs becomes of crucial importance. Singh and Reddy [17] used metrics like scalability, fault tolerance, and data I/O rate to compare platforms for big data processing, but their work was focused on hardware platforms and only on a few software frameworks like Hadoop [18] and Spark [19] that are used for batch processing. In the past years, batch processing of big data was a focal point in distributed processing.

Nevertheless, numerous applications like environmental monitoring and fraud detection applications require continuous and timely processing of information. The area of stream processing is not new. In 2005, Stonebraker et al. [4] defined the requirements that should be met by a real-time stream processing system to handle stream processing applications. Consequently, several attempts were dedicated to present solutions provided by the community in stream processing. Examples of early stream processing frameworks like Esper, Aurora and Borealis are discussed in [20]. The term Information Flow Processing (IFP) systems was used to describe systems providing continuous and timely processing of information. Authors present different aspects of these systems and provide a useful model for their classification. Nevertheless, the term 'big data' and the challenges arising by it are not mentioned at their work. With the emergence of IoT in recent years, new distributed stream processing systems arose.

Research and developments in big data stream processing are ongoing and of great interest against the challenges posed by business trends. Brief reviews over initial solutions [10,21,22] and comparisons based on the basic selection criteria such as language support and documentation [11] soon led to full surveys over a number of available choices and valuable comparisons between them for potential users [12,13,14,15,16]. Hesse and Lorenz [12] presented prominent stream processing engines and key components of their architectures, and compared them on performance metrics like latency, throughput and message processing assurance. Details on systems' mechanisms leading to specific performance characteristics are not, though, provided. Authors praised the need for benchmarks and scalability analysis in their work.

Towards this direction, Georgiadis [13] provided a useful overview of the frameworks studied by Hesse and Lorenz [12] and tested the way two of them, Spark and Flink [23], exploit cluster's resources as they scale out, using an appropriate benchmark. Details on aspects like memory and iteration management were also studied for these frameworks. Efficient resource management is also a matter of interest for Assuncao et al. [14]. The authors provided a survey on stream processing engines and mechanisms for exploiting resource elasticity features of cloud computing proposed in the literature, were also examined. Assuncao et al. focused on the mechanisms required to exploit the elasticity features of a stream processing engine and the comparisons between various techniques focus on the metrics considered when scheduling a scaling operation.

Kamburugamuve et al. [15] evaluated streaming solutions on their programming API for developing applications and their execution model. To compare the stream processing frameworks they divided them into separate layers and evaluated the functionality of each layer. Authors support that aspects like fault tolerance and high performance of the execution engine raise the need for research on scheduling of streaming tasks. In-built scheduling techniques in most processing systems mentioned in the aforementioned literature are far from optimal but most systems allow users to specify custom scheduling for tasks.

In this work, we compare and evaluate the stream processing systems by first specifying the key mechanisms of each system and by showing how these mechanisms lead to certain performance features. As far as the practical part is concerned, a few experiments that investigate the performance and the impact of fault tolerance mechanisms used by several stream processing systems have been conducted [24,25,26,27,28] and are presented in our work. More real-world streaming experiments would be useful for potential users as most of the available systems are frequently updated by their communities, new possibilities are added, and new engines arise to replace them. Also, we present several scheduling approaches in stream processing, discuss scheduling techniques being developed, and outline future directions.

3. Our approach in data acquisition and selection criteria

To address the research questions as outlined in the Introduction, we surveyed the literature between January 2010 and March 2018. The choice of the review period was a practical one and took into consideration the fact that big data is a rather recent phenomenon. There is only one exception. Stonebraker's et al. paper 'The Requirements of Real-Time Stream Processing ,' goes back in 2005 but valued as a crucial prototype that still characterises modern stream processing frameworks in the big data industry and we tried to present it adapted it in this context.

The bibliographic analysis in the domain under study involved three steps:

(1) collection of related work;
(2) filtering of relevant work;
(3) detailed review and analysis of the state of the art.

Steps had to be repeated both for Sections 4 and 5. In the first step, a keyword-based search for conference papers, journal articles, technical reports was performed using the following databases: IEEE Xplore, Science Direct, ACM, and CiteSeer. These databases were chosen because of their wide coverage of relevant literature and advanced bibliometric features such as suggesting related literature or citations. Keywords like 'stream processing,' 'real-time ,' 'big data,' 'cloud computing,' 'distributed processing,' were used, in possible combinations. After a literature of the state of the art had been studied, we proceeded with bibliographic search based on keyword combinations using a search engine that indexes the full text and metadata of scholarly literature, Google Scholar to add more specific information. Terms like 'windows,' 'watermarks,' 'recovery,' and 'state management' were added to our 'pool' of keywords to explore the mechanisms used in big data era to handle requirements in stream processing.

To select the major solutions that satisfy stream processing requirements and perform comparison between them, we consulted surveys [10,11,12,13,14,15,16,21,22] presented in the 'Related work' Section. We had to select a subset of the state of the art systems, which are able to satisfy the requirements and use the most representative mechanisms to do so. We also wanted to focus on open-source solutions with active community support that would also be useful for the second part of our work referring to task scheduling.

Initially, we chose two main representatives of the two available execution models for processing streams to understand how they perform; Apache Spark Streaming [29] and Storm[30]. Spark is the most active project in terms of community numbers and utilises the micro-batch technique while Storm uses a streaming computation model (Heron [31] is an improved version of Storm but it is not such a mature project and we would not be able to find enough work addressing scheduling techniques on it). Flink [23] can support the same capabilities as Spark but works as a native stream engine and arises several challenges for Spark in stream processing (e.g. in case of recovery and latency). Thus, it was added to our review. Flink also seems to have more capabilities than Storm. Nevertheless, Storm has a larger community and as it is a mature project, there is extensive literature on scheduling techniques on it. Samza [32] is a distributed stream processing framework that is interesting chiefly due to the capabilities arising by its smart coordination with Kafka message broker [33]. This provides several advantages like automatic flow control built in the system. Consequently, we ended up with four prominent stream processing frameworks namely Apache Storm, Spark, Samza and Flink.

The names of frameworks under consideration; 'Spark Streaming,' 'Storm,' 'Samza,' and 'Flink' were added to our keywords 'pool' and used in several combinations to check how the issues studied are addressed by prominent frameworks, provide an overview, and make comparisons between them based on a list of criteria. For Section 5 keywords 'task scheduling,' 'stream scheduling,' and 'resource allocation' were used in combination with most of the aforementioned keywords to retrieve scheduling approaches either incorporated in the existing frameworks or proposed with new possible architectures.

In the second step, we restricted our study to a number of papers having the highest quality and considered as the most important resources in this field. To do this, we focused on papers with higher impact (that is, higher number of citations). An exception was made for papers published in 2016–2018, where papers with a small number of citations were also accepted. In this way, gray literature, though quite informative for big data industry conditions, was avoided. Nevertheless, official documentation of each framework complemented the related articles retrieved several times.

We further filtered out papers referring to matters of interest (e.g. distributed processing, stream processing, scalability, etc.) in the cloud that did not make actual use of big data and their inherent characteristics (i.e. volume, velocity, variety). We also had to filter out papers not referring to stream processing of big data, as literature is much more rich when it comes to processing of already stored big data and Hadoop MapReduce [34] scheduling enhancements. In this way, the number of papers qualified according to our constraints was severely restricted.

In the last step, we studied the selected literature in detail to extract the information relevant to our research questions. Additional literature that had not been identified in the first step was retrieved in this step as well if they were referred to by 'related work.' This 'snow-ball' approach resulted in additional articles and web-items from which relevant information was extracted as well. The extracted information was analysed and synthesised.

4. Stream processing of big data

Centralised computing systems have been around in technological computations for years. In such systems, one central computer controls the peripherals and performs complex computations. Centralised computing systems require expensive hardware in order to process huge volumes of data and support multiple online users concurrently. Under these circumstances, cloud and distributed computing systems arose to exploit parallel processing technology. Users can share computing resources, which can be virtualised and allocated dynamically in the cloud, typically characterised by scalable and elastic resources.

Nowadays, distributed cloud computing attract numerous customers and service providers. The importance of stream processing systems increases as more and more modern applications impose tighter time constraints on a particular event. The resultant analysis provides information that can provide companies with visibility into many aspects of their business and customer activity and enables them to respond rapidly to emerging situations. Much of the data that companies receive in real time is more valuable at the time it arrives. For example, businesses can track changes in public preference on their brands and products by continuously analysing social media streams. There is no point in detecting a potential buyer after the user leaves the e-commerce site or detecting a credit card fraud after a transaction is completed [35]. For instance, eBay detects frauds from PayPal usage by analysing 5 million real-time transactions every day. Online machine learning is also another promising use case. There are numerous use cases making the need for stream processing imperative. When data arrives fast and need to be processed with real-time restrictions, a stream processing system has usually to be placed on top of a cloud infrastructure. Processing and analysing data streams in real time are tasks that traditional data-warehousing environments cannot handle easily due to high latency and cost [36].

A stream processing system or data stream management system (DSMS), is designed to handle data streams and manage continuous queries. It executes continuous queries that are not only once performed, but are continuously executed until they are explicitly uninstalled. It produces results as long as new data arrives in the system and data is processed on the fly without the need for storing it. Data is usually stored after processing. Stream processing systems differ from batch processing systems, due to the requirement of real-time data processing. The term 'real-time processing system' refers to a system that responds within 'real-world' time deadlines. It guarantees that a certain process will be executed within a given period, maybe a few seconds, depending on the quality of service constraints.

The term 'real-time' is a bit redundant but many systems use the term to describe themselves as low latency systems. Elaborate and agile systems have been proposed for these new demands.

Streaming applications can be represented using a directed acyclic graph (DAG), where vertices are the operators and the edges are the channels for dataflow between operators [12]. The processing can go through operators in a particular order, where the operators can be chained together, but the processing must never go back to an earlier point in the graph. There are two execution models used in stream processing [37]:

1. *The Stream-Dataflow Approach*, where an application is viewed as a dataflow graph with operators and data dependencies between them (sometimes referred as operator-based approach). A task encapsulates the logic of a predefined operator like filter, window, aggregate or join or even a routine with user-specified logic. A data stream between two operators represents an infinite sequence of data produced by a task, which is available for further consumption. Data is delivered and consumed in arbitrary order across parallel tasks, and as a result, there is a lack of a coarse-grain unit for transactional processing. Everything is automatically pipelined.
2. *The Micro-Batch Approach*, that offers a solution to enable processing data streams on batch processing systems. With micro-batching, we can treat a streaming computation as a sequence of transformations on bounded sets by discretising a distributed data stream into batches, and then scheduling these batches sequentially in a cluster of worker nodes.

The progress of real-time stream processing systems in the cloud has been relatively slow, but nowadays there are several solutions offered. Apache Storm [30] is a real-time distributed computing technology for processing streaming messages on a continuous basis. Individual logical processing units are connected like a pipeline to express a series of transformations and expose opportunities for concurrent processing. Heron is a processing engine for streaming and real-time data at a scale that was developed at Twitter as a replacement for Apache Storm. Spark Streaming [29] is another solution that makes it easy to build streaming applications using the micro-batching approach. The idea behind this is to process in the same fashion as the batch processing but keeping the batch sizes very small. Apache Samza [32] is a distributed stream-processing framework that provides a simple API, comparable to MapReduce. There are also commercial solutions like Amazon Kinesis [38], a fully managed service for real-time processing of streaming data at massive scale and IBM InfoSphere Streams [39]. Research and development in the area of stream processing is continuous and becomes of great importance in the mentioned context of IoT. In the following section, we are going to outline issues and requirements that stream processing systems have to meet to excel at real-time stream processing applications. Our work relies mostly on Stonebraker's et al. [4] analysis. Also, we present the mechanisms used in the big data era to face the aforementioned challenges.

4.1. Issues/requirements in big data real-time stream processing

Stonebraker et al. [4] introduced the requirements for real-time processing of data streams to provide high-level guidance of what to look for when evaluating stream processing solutions. The most prevalent of them can characterise the available stream processing frameworks for big data in ways that are presented below and could be further enhanced. A real-time stream processing system has to:

1. '*Process messages in-stream without any requirement to store them to perform any operation or sequence of operations*.'
   More and more modern applications impose tighter time constraints on a particular event. Performance related requirements like latency and throughput are extremely important in streaming applications. To meet these requirements, processing has to be done without the costly storage operation. MapReduce can handle large datasets but works using permanent storage [34]. Thus, it fails when it comes to real-time data processing, as it was designed to perform batch processing

on voluminous amounts of data. In-memory computing provides a solution to this problem and is based on using a distributed main memory system to store and process big data in real time. Main memory delivers higher bandwidth and better latency compared to hard disk. Even if the framework uses memory for caching of frequently used data, the whole job execution performance will be improved significantly [21]. Despite the dropping price of memory, using large amounts of RAM to run everything in-memory can be expensive, so proper mechanisms are needed to handle it in an effective way.

Spark Streaming is a system that arose to provide in-memory computation effectively. It processes all data in-memory, only interacting with the storage layer to initially load the data into memory and at the end to persist the final results. To implement in-memory computations, Spark uses a model called Resilient Distributed Datasets or RDDs, its in-memory abstraction to work with data. Flink also offers in-memory computations. Its core is built on a data flow streaming engine whose fundamental functionality is pipelining i.e. all tasks have to be online simultaneously in order for the data to be able to flow through the tasks regardless of the node, that a task might reside on [13]. Storm runs also in-memory to process big data at in-memory speed. Memory is a resource of crucial importance for a DSMS. Since in-memory computing is not only about storing but also processing big data in real-time, problems like efficient scheduling and appropriate moving code to data rather than data to code are matters that it has to deal with [21].

2. '*Support a proper querying language with extensible stream-oriented primitives.*'

Stream processing systems receive input streams from one or more sources and organise the computation into a directed graph of operators either explicitly or implicitly. In the first case, systems are known as compositional. They offer basic building blocks for composing custom operators and topologies. The user has to implement the whole logic himself and the operators are defined as implementations of classes. In the latter case, systems are called declarative and developers are provided with high-level languages that are automatically translated by the system into the operator graph [37,15].

Querying mechanisms are needed to detect events of interest or compute real-time analytics. General purpose languages like Java have been used as programming tools in streaming applications but using low-level programming schemes results in long development cycles and high maintenance costs. Using a stream processing system with no support for SQL like query languages requires sound knowledge on imperative style programming and distributed systems to effectively utilise it. Support for SQL like continuous query languages or SQL with streaming extensions can help towards this direction. SQL remains a reliable query language with high performance for real-time analytics [40] but has limitations when dealing with huge amounts of data. Systems can adopt SQL-like languages that represent the processing as queries that get repeatedly and continuously evaluated as new data becomes available. New projects like SQLStream [41] and Apache Calcite [42] emerge to execute queries over big data using a set of streaming-specific extensions to standard SQL. There have been a number of efforts towards using various languages in event processing systems to allow users to express their interests. Different projects have taken different directions with the research [43].

There is a concept of vital importance when it comes to data stream mining. The data set is assumed to be infinite, creating problems in processing. Not all operators can be evaluated over streams. A stream processing engine has to know when to finish an operation on a stream of data and output an answer. Traditional methods have the advantage of knowing the total size of the set. Sampling is a tool that addresses this problem. Windowing is a sampling method defining the scope of an operation and is a heavily used approach in stream processing. An unbounded stream of data is split into finite sets, called windows, based on specified criteria, like time. A window can be conceptualised as an in-memory table where events are added and removed and computations are calculated on each window of events. In [44,45] there are detailed analysis over windowing techniques. There are stream processing frameworks that offer only the basics, like Storm, Samza and Spark Streaming that only supports time windows having a size and slide that

are multiple of the batch size. On the contrary, Spark Structured Streaming, a new component in Spark 2.0, offers more windowing possibilities. Flink also offers a wide range of windowing features. Tumbling windows, sliding windows, session windows and global windows are pre-implemented while users can also implement their own windows. The notions of windowing and time, and the windowing semantics used by prominent stream processing frameworks are further discussed in [46].

3. *'Use mechanisms to provide resiliency against stream imperfections including out-of-order or missing data which are commonly present in real-world data streams.'*
Incomplete datasets, destroyed data, the presence of outliers or biases in the training affect the analysis' accuracy. Missing data is a phenomenon that is inevitable in distributed communication environments. A transmission loss due to broken link between sensors or a malfunctioning sensor leads to missing values from the data. A sensor accident can lead to permanent missing values while a temporal disconnection or network delay leads to temporal loss as data may arrive in a short while [47].

   The assessment of data quality demands significant human involvement and expert knowledge but when it comes to large volumes of unstructured data even semi-automated approaches are not practical. Streaming data and real-time processing outweigh data quality, making data quality management more imperative than ever. To the best of our knowledge, the above problems have not been addressed effectively so far, but there is growing interest towards this direction [47,48]. Techniques like outlier detection, data transformations, dimensionality reduction, cross-validation and bootstrapping are valuable tools in data quality management but until recently, data quality research has primarily focused on structured data stored in relational databases and file systems.

   Streaming data is also typically not well ordered in time. Event-time ordered data is uncommon in many real-world, distributed input sources. While receiving a stream of an IoT sensor readings, for example, some devices might be offline, and send data after some time. Leaving a strictly time-constrained system waiting is not an appealing solution. Keeping all windows around forever would also consume all available memory. When it comes to real-time processing, there must be a mechanism to allow windows to stay open for a while. Watermarks are such a mechanism. They enable streaming systems to emit timely, correct results when processing out-of-order data. They are used as a heuristic, assuming that all events before a specific time have been observed. This technique is further explored in [45]. Most prominent stream processing systems (e.g. Spark Structured Streaming, Storm, Samza, Flink etc.) implement watermarking techniques.

4. *'Ensure that the apps are up and available and the integrity of the data should be maintained at all times despite failures. The system should have the capability to efficiently store, access state information and combine it with live streaming.'*
Network failures, lack of resources, and network software bugs can cause even more problems in large-scale distributed computing. When large sets of such components are working together there is a high probability that at least one component may fail at a given time. Almost all stream processing systems in big data industry provide the ability to recover automatically from faults. Several techniques have been developed to recover fast enough so that the normal processing can continue with the minimal effect to the overall performance. We distinguish three main categories for recovery in stream processing; precise recovery, rollback recovery and GAP recovery (the interested reader can refer to [49] for a thorough analysis of fault tolerance in stream processing engines). Precise recovery provides no evidence of a failure afterwards but there is an increase in latency. In rollback recovery, the output produced after a failure is 'equivalent' to, but not necessarily the same as, the output of an execution without failure. For example, information may be processed more than once when a failure happens. In GAP recovery the loss of information is expected in favour of reduced recovery time and runtime overhead.

   Storm and Samza use upstream backup techniques that provide roll-back recovery. Generally speaking, a stream processing system can use the upstream backup method to avoid any

checkpointing overhead e.g. disk I/Os and data structures (checkpointing is also available in both aforementioned systems). When it comes to state, though, Storm requires the user to manually handle recovery of state [50]. Apache Zookeper [51] has to be used to maintain its cluster state while Samza makes state changes fault tolerant by modelling them as an output log (commitlog or changelog) to a Kafka topic [33]. Systems like Spark Streaming that work with batches usually re-execute the necessary computations in case of failures, yielding the same output regardless of them due to RDDs, that can be recomputed deterministically. RDDs are immutable, meaning that no worker node can modify it, it can only process it and output some results. Each RDD can trace its lineage back through its parent RDDs and ultimately to the data on disk [52,53]. Flink implements different recovery mechanisms; rollback/restart for finite streams and distributed snapshotting for infinite streams. Flink's checkpointing mechanism stores consistent snapshots of distributed data stream and operator state. These fault recovery methods lead to different processing guarantees for each system (and provide fault tolerance in different ways) that are further examined in Section 4.2.

5. *'Be able to distribute its processing across multiple processors and machines to achieve incremental scalability. Ideally, the distribution should be automatic and transparent.'*
   Streams do not have preset lifespan, arrive at non-predefined rates and are voluminous. If not managed carefully, processing delays can become unacceptable and lead to long queues at a processing node, buffer overflows, and memory exhaustion. The number of stream computations is much more than the number of machines available for processing. Multiple stream processing computations should be interleaved on the same machine to reduce the number of needed connections and assure high throughput and increased performance. On the other hand, if tasks are densely allocated in nodes, a node failure will bring down most of the application. Overloaded and underutilised machines should be avoided as imbalances deteriorate system's performance.

   Running algorithms in a sequential manner is not sufficient. Algorithms should run in parallel with streams of data also partitioned to many distributed processing units. Moreover, unlike stateless computations, stateful computations cannot simply be replicated on multiple machines with streams of data been processed in e.g. a simple round-robin fashion in parallel. Horizontal scaling requires adapting the graph of processing elements, exporting and saving operators' state for replication, fault tolerance and migration [14]. As stream processing queries are often treated as long running that cannot be restarted without incurring a loss of data, the initial task assignment, where processing elements are deployed on available computing resources becomes more critical than in other systems. Processing must be orchestrated carefully over a collection of machines making task scheduling either statically or dynamically at scale over a set of machines, a challenge. This led us to investigate scheduling techniques in the streaming big data era in Section 5.
6. *'Have a highly optimised, minimal-overhead execution engine to deliver real-time response for high-volume applications.'*
   The aforementioned rules make no sense alone unless an application can process high-volumes of streaming data with very low latency and high throughput, to meet the real-time demands. The ability to process large volumes of data on the fly, as soon as they become available, is a fundamental requirement in today's information systems. There is a rapid increase in the number of available stream processing engines. However, each engine defines its own processing model and execution semantics that affect its performance. For instance, Apache Storm uses the dataflow execution model where streams of data are processed tuple by tuple on continuous operators. On the other hand, Spark streaming creates small batches of streaming data and execute them based on its batch processing engine. To have this requirement met, it is imperative that any user with a high-volume streaming application carefully examine the given solutions to the one that better fits his use case and its demands. We provide a comparison between the systems under study in Section 4.2 and this comparison's results in Section 4.3 to support potential users towards this direction.

Taking into account the aforementioned requirements, the nature of streams poses several processing challenges. The matters of performance and message processing assurance seem to rise to the top when choosing a framework for a specific application. We are going to further examine the aforementioned DSMSs with respect mainly to these matters to further check and compare their capabalities and the results of their built-in mechanisms. Efficient task scheduling strongly affects both handling fault tolerance and performance [15]. Most streaming systems allow the user to specify custom scheduling for tasks and the interest on this aspect grows widely but there is still much work to be done. This raised our interest to examine the problem of task placement and resource allocation in a distributed setup in Section 5.

4.2. Comparing and evaluating stream processing solutions

Available frameworks have several differences in their architecture and in the processing model they use. Choosing a system that can guarantee fault-tolerant, high-performance stream processing based on user's needs is quite cumbersome. The mechanisms overviewed to handle the stream processing requirements provide different functionality characteristics to each DSMS. In this section, we divide the functionality of a streaming engine mainly in matters in the context of performance and fault tolerance to evaluate the selected DSMSs. Related surveys [10,11,12,13,14,15,16] were also consulted for defining the list of criteria used for our purposes.

All streaming solutions examined in this section use a parallel distributed architecture that allows portioning of data streams and parallelisation across a cluster of machines. The attributes that are also examined are the following:

Processing Model: The selection of a processing model for a system varies from batch processing and micro-batch processing to stream processing tuple by tuple. Batch processing systems such as MapReduce are out of our interest in this section.

Stream Primitive: Refers to the main data structure in a streaming system. These systems use various words for such concepts.

Latency: Refers to the elapsed time from job submission to receiving the first response.

State Management: Streaming computation can be either stateless or stateful. 'A stateless programme looks at each individual event and creates some output based on that last event' [54]. For example, a streaming programme might receive traffic data and raise an alert in the event of traffic light violations. 'A stateful programme creates output based on multiple events taken together' [54]. All kinds of state machines used for complex event processing (CEP) are stateful. For example, creating an alert after receiving two traffic light violations that differ by less than 5 min is a stateful computation. The frameworks examined use various strategies to store states or may not store state.

Throughput: Refers to the average number of jobs or tasks or operations per time unit.

Delivery Guarantee: Refers to the 'level of correctness' of the results produced after a failure and a successful recovery of the system compared to what the results would be without any failures. Stream processing systems are characterised by the following three semantics [32]:

- *at-most-once delivery*, which drops messages in case they are not processed correctly, or in case the processing unit fails. This is usually the least desirable outcome as messages may be lost.
- *at-least-once delivery*, which tracks whether each input was successfully processed within a preset timeout. It this way it guarantees that messages are redelivered and re-processed after a failure. If a task fails, no messages are lost, but some messages may be redelivered. In case the effect of a message on state is idempotent, no problem occurs if the same message is processed more than once. A duplicate update will not change the result. However, for non-idempotent operations such

as counting, at-least-once delivery guarantees can give incorrect results. This approach is good enough for many use cases but it may cause duplicates.
- *exactly once semantics*, which uses the same failure detection mechanism as the at-least-once mode. Messages are actually processed at least once, but duplicates can be avoided via various techniques. Such systems guarantee that the final result will be exactly the same as it would be in the failure-free scenario. This is the most desirable feature but it is difficult to guarantee in all cases.

Programming Model: Systems can be either compositional when users have to model the streaming application as graph explicitly or declarative when users are provided with higher level abstractions.

API languages: Refers to the languages that someone can use in order to develop an application for this framework.

Contributers: Refers to the respective community of contributors based on Github.

Table 1 summarises how four different frameworks support the above features.

Storm implements the dataflow model. The topology of an application is described by using components named 'spouts' and 'bolts,' referring to data sources and elements that process data in the form of tuples respectively [55]. Since a tuple is processed as it arrives, Storm has sub-second latency. Its mechanism supports low throughput mainly because of its acknowledge mechanism. Each record that is processed from an operator sends an acknowledgement back to the previous operator, indicating that it has been processed. This mechanism may also falsely classify a number of records as non-aknowledged. Therefore, these records will have to be re-processed leading to even lower throughput. It does not provide state recovery but provides guaranteed delivery and processing of data using the upstream backup mechanism and record acknowledgements [30]. In case of failure (e.g. worker failure), if not all acknowledgements have been received, the records are replayed by the spout [24]. In this way, no data loss will occur but duplicate records may pass through the system. Mutable states may be incorrectly updated twice. That's why it offers at-least-once delivery guarantee. Micro-batch processing is offered by Trident Storm for higher throughput. In Storm Trident the state can be managed automatically, so it does guarantee state consistency (exactly once delivery) but state is kept in a replicated database which is expensive, as all updates are replicated across the network [50,56]. Apache Storm is a compositional system that expects user to explicitly define the application DAG.

Spark Streaming batches up events within a short frame before processing arrived data. At the low-level, data is represented as RDDs and computations on these RDDs can be represented as either transformations or actions. The abstraction for data streams is called Dstream and it consists of an RDD sequence, containing data of a certain stream interval. DStreams let users apply transformations to them.

Streaming computations in Spark Streaming, represent a series of batch computations of a definable time interval size [57]. Adapting the batch methodology for stream processing involves buffering the data as it enters the system leading to few seconds latency. There is a penalty of latency equal to the micro-batch duration. Waiting to flush the buffer also leads to an increase in latency. Nevertheless, the buffer allows Spark Streaming to handle a high volume of incoming data, increasing overall throughput [16]. As we can see there is a trade-off between low latency and high throughput. Systems based on micro-batching can achieve high throughput but in case processing of a batch takes longer in downstream operations than in the batching operations, the micro-batch will take longer than configured. This may lead to more and more batches queueing up (or to a growing mini-batch size) [33]. While Spark Streaming may be adequate for many projects, it is not a true real-time system. Zaharia et al. [58] mention, though, that while this model is slower than true streaming, the latency can be minimised enough for most real-world projects due to the use of RDDs. RDDs allow in-memory computations in a fault-tolerant manner, avoiding writing outputs to replicated, disk storage systems yielding to time-consuming disk I/Os. Spark Streaming provides support for both stateful and stateless computations and guarantees that batch level processing will be executed in an exactly once manner.

Table 1. Classification of streaming solutions.

Platform/criteria	Processing framework	Stream primitive	Latency	Throughput	Stateful operations	Guarantee	Programming model	API languages	Contributers (Github)
Storm	Streaming	Tuple	Subsecs	Low	No	At least once (exactly with Trident)	Compositional	Any	294
Spark streaming	Micro-batch	Dstream	Few Secs	High	Yes	Exactly once	Declarative	Java, Scala, R, Python	1287
Samza	Streaming	Message	Subsecs	High	Yes	At least once	Compositional	JVM languages	93
Flink	Hybrid	DataStream	Subsecs	High	Yes	Exactly once	Declarative	Java, Scala, Python	444

This is achieved by tracking the lineages in each DStream. All state in Spark Streaming is stored in RDDs [58]. It is a declarative system as it introduces several abstractions for representing data and managing different types of computations. The code defines just the functions that need to be performed on the data and Spark implies the corresponding DAG from the functions called.

Samza is a stream-processing framework based on the Publish/Subscribe model. It listens to a data stream, processes messages which are its stream primitive as they arrive, one at a time, and outputs its result to another stream. It is tightly tied to the Apache Kafka messaging system [33] for streaming data between tasks and Apache YARN [59] the for distribution of tasks among nodes in a cluster. Kafka offers replicated storage of data that can be accessed with low latency, so Samza jobs can have latency in the low milliseconds when running with it. Samza allows tasks to maintain state by storing it on disk (typically using Kafka). This state is stored on the same machine as the processing task, to avoid performance problems. By co-locating storage and processing on the same machine, Samza is able to achieve high throughput [32]. It also tracks whether a message is delivered or not and it redelivers it in case of failure, to avoid data loss using a checkpointing system but can only deliver at least once guarantees. If a Samza task fails and is restarted, it may double-count some messages that may have been consumed since the last checkpoint was written. The topology of a Samza job is explicitly defined by the user's code [15].

Flink is an hybrid solution [60]. The need to manage different workloads under a coherent architecture led to several design patterns with the most popular being the 'Lambda Architecture.' The complexity of using different batch and streaming architectures paved the way to the 'Kappa architectural' pattern that fuses the batch and stream layers together. Flink is a materialisation of the Kappa architecture. Despite the fact that it relies on a streaming execution model, it is possible to process both bounded and unbounded data, with two APIs running on the same distributed streaming execution.

The basic data abstraction for stream processing is called DataStream. It executes arbitrary dataflow programmes in a data-parallel and pipelined manner, which results in achieving low latency. Apache Flink's dataflow programming model provides event-at-a-time processing [61]. Tuples can be collected in buffers with an adjustable timeout before they are sent to the next operator to turn the knob between throughput and latency. It performs at large scale, running on thousands of nodes with very good throughput and latency characteristics based on existing benchmarks. When using stateful computations, it ensures exactly once semantics. Apache Flink includes a lightweight fault tolerance mechanism based on distributed checkpoints. Its algorithm is based on a technique introduced by Chandy and Lamport [62] and periodically draws consistent snapshots of the current state of the distributed system without missing information and without recording duplicates. These snapshots are stored to a durable storage. In case of failure, the latest snapshot is restored, the stream source is rewinded to the point when the snapshot was taken, and is replayed [23]. Flink is currently a unique option in the processing framework world but at the moment, it is a young project and there hasn't been much research into its scaling limitations. It is a declarative system, providing higher level abstractions to users like Spark. The DAG is implied by the ordering of the transformations while its engine can reorder the transformations if needed.

4.3. Comparison results

The provided overview reveals that there is no system able to do everything and no system that does nothing. All systems do something but they do it differently. Benchmarking can be a good way to compare them, especially when it has been done by third parties. To shed some light on the performance of the above engines, a few experiments that investigate latency, throughput and the impact of their fault tolerance mechanism have been conducted. Some examples are provided below:

- Cordova [24] compared Spark Streaming and Storm Trident. He provided a benchmark of both systems over different tasks and processing tuples of different sizes. The main conclusion was that

Storm Trident was around 40% faster than Spark, processing tuples of small size. However, as the tuple's size increased, Spark had better performance maintaining the processing times.

- Chintapalli et al. [25] designed and implemented a real-world streaming benchmark focusing on Storm, Flink and Spark Streaming and found that Storm and Flink have much lower latency than Spark Streaming at fairly high throughput. On the other hand, Spark Streaming is able to handle higher throughput and its performance is quite sensitive to the batch duration setting.
- Perera et al. [26] compared Flink with Spark against two benchmarks, Intel HiBench Streaming and Yahoo Stream. Both systems performed similarly under different loads but Flink demonstrated a slightly better performance at lower event rates, mainly because of Spark Streaming's micro-batch technique. The CPU utilisation was similar in both systems but when it came to memory usage Flink needed less amount of memory than Spark.
- Lu et al. [27] also proposed a benchmark definition named StreamBench. They applied StreamBench to Spark Streaming and Storm. They found that Spark tends to have larger throughput (even about 5 times that of Storm's) and less node failure impact compared to Storm. Storm, though, has much lower latency (even 50 times less) than Spark, except with complex workloads under large data scale, for which its latency may be multiple times of Spark's. A mediator system was used between the data source and the streaming system to decouple them.
- Karimov et al. [28] proposed a benchmark framework in which they separated the system under test from the driver to avoid mediators and conducted experiments with Storm, Spark and Flink. Both Flink and Spark were robust to fluctuations in the data arrival rate in aggregation workloads while for join queries, Flink behaved better. In case latency is a priority, Flink seemed to be the best choice and had better overall throughput when tested in aggregation and join queries.

To provide accurate and customised recommendations, more real-world streaming benchmarks on the aforementioned systems based on different use cases need to be designed and implemented. Nevertheless, we can't completely rely on benchmarking in stream processing as even a small change in configuration or use case can completely change the numbers. Moreover, the above technologies may have established themselves as leaders and have strong supporting communities, but are still evolving. For instance, most of the aforementioned benchmarks do not take into consideration the release of Spark Streaming 2.0 that overcomes several limitations with RDDs improving its performance towards other systems. Consequently, it is not easy to make a clear ranking with quantifiable results. Combining the above overview (Table 1) with available benchmarks, though, some conclusions, arise:

- Storm works with very low latency but can deliver duplicates and cannot guarantee ordering by default configuration. Since it does not provide implicit support for state management, it does not fit in cases of complex event processing. Nevertheless, it excels in case of non-complicated streaming use cases, where latency is of crucial importance, due to its true streaming nature and maturity.
- Samza has to be integrated tightly with Kafka and YARN to provide high performance, flexibility, and state management. It is not easy to be used without them in the processing pipeline, while deploying such a system would require extensive testing to make sure that the topology is correct. Nevertheless, in case these technologies are already incorporated, it is a mature, fault-tolerant solution providing high performance (both in matters of latency and throughput). At least-once processing guarantee remains an issue but it is referred that is about to be resolved in the next release.
- Spark is very popular, mature and widely adopted with a strong community supporting it. It provides high throughput, it is fault tolerant because of its micro-batch nature providing exactly one guarantee, and can be used in case sub-latency is not required. Nevertheless, it lags behind Flink in many advanced features. Its new release (Structured Streaming) is equipped with several good features and promises to yield subsecond latency, but it's early for this ambition to be achieved.

- Flink provides true stream processing with batch processing support. It is heavily optimised, incorporating several innovations like light weighted snapshots and it seems that it's the leader in the DSMS landscape. As we saw, most of the wanted aspects (low latency, high throughput, exactly once guarantee, state management) are provided. Nevertheless, there was a lack of adoption initially and its maturity is a matter of concern. Its community support is also smaller than Spark's. This seems to change rapidly, though, as it started to get widely adopted in the business world.

Consequently, the best possible answer provided to users that want to learn which is the best possible solution is that it depends on their needs. Future considerations should also be taken into account. For instance, in simple cases Storm may seem a good idea. Nevertheless, in case advanced requirements involving complex event processing like aggregations and joins occur later, or batch-oriented tasks should also be implemented, advanced streaming frameworks like Spark Streaming or Flink should be preferred. Changes in existent infrastructure and re-training employees may lead to huge costs in time and money.

Finally, we should keep in mind that coding in declarative systems is much easier than in compositional as users are provided with higher level abstractions and imply the DAG through their coding. Optimisations can be done by the system. Nevertheless, in compositional systems, the code is at the complete control of the developer. If there is a need for fast and easy implementation, systems like Spark or Flink should be preferred but if complete control over the application's graph is needed, then Storm or Samza should be chosen.

As we see, the best fit for each situation depends upon several factors. Understanding the mechanisms and characteristics of the aforementioned architectures makes it easier to pick or at least filter down the available options. Work-in-progress evaluation can then support users to make the best possible choice based on their needs. Changes in configuration, tuning or available infrastructure can change results and lead to severe improvements. Task scheduling strongly affects the performance and fault tolerance in a stream processing system and is going to be examined in the next section.

5. Scheduling approaches in stream processing

The aforementioned frameworks use special data processing mechanisms and architectures to provide the necessary performance. The efficient resource allocation and task scheduling of streaming applications are of great importance in data stream processing. *Resource allocation* deals with the problem of gathering and assigning resources to different requests. Inside an application, resources need to be carefully scheduled to different components to ensure optimal utilisation. Misplacing resources can cause both poor resource utilisation and instability of the system as a whole. Overprovisioning resources to each operator can lead to resources waste, while load shedding leads to incorrect results [63]. *Task scheduling* controls the order of job execution and cares about which tasks and when to be placed on which previously obtained resources [64]. Appropriate scheduling techniques can improve system's performance and reduce energy consumption and costs, especially when we move to the cloud. However, as we have already seen, goals like low latency and high throughput are often in conflict. Thus, a scheduler has to implement a suitable compromise, depending upon the user's needs and objectives.

As far as task scheduling is concerned, queries are passed to the data and Hadoop has the ability to move the computation close to where the actual data is placed, instead of moving large data to computation. This minimises network congestion and increases the overall system's throughput. Data locality plays a role of crucial importance. Nevertheless, MapReduce is not effective on processing huge volumes of data in real-time. It is a framework that is not efficient in executing interactive jobs and real-time queries.

Scheduling approaches and resource allocation for real-time stream processing of data basically differ from traditional batch processing. It is much more difficult due to the dynamic nature of input data streams. Unlike batch processing systems, stream processing systems are designed to process

unbound streams of data. A job in the first case will eventually finish, while a job in the latter continues forever, unless the user kills it. The objective of stream processing is to build a reliable, high-throughput and low latency event processing that can continuously analyse and act on real-time streaming data. In batch processing systems, computations are assigned to the nodes where the required data is stored, while in streaming most communicating tasks are placed together on one node or rack. Moreover, in memory computations used in stream processing systems, are hundreds of times faster than computations running on a disk as is traditional for systems like Hadoop.

Stream processing systems following the micro-batch approach have several advantages over stream processing systems that process data by one record at a time, like fast recovery from failures, better load balancing and scalability. Micro-batch systems are optimised for throughput but have increased query response time, since each input has to wait until a batch is formed. Extremely small batches could possibly minimise this extra latency, but this would cost extra overhead. On the other hand, operator-based stream processing works better when we have to deal with strict real-time constraints but is prone to faults [63]. To provide the necessary system performance at high load incoming data, additional processing mechanisms and efficient scheduling of streaming applications are needed.

We divide scheduling decisions in offline (using static algorithms) and online (using dynamic algorithms). Algorithms may rely on predefined characteristics of streaming topologies (offline) or may gather information by monitoring systems (online). Offline decisions rely on the knowledge a scheduler has before any task is placed and running. Online decisions refer to information gathered during the execution of user's application, after the initial placement of its tasks over the cluster's nodes. We are going to further examine and categorise works proposing heuristic algorithms (or sometimes even whole architectures) that try to choose the optimal placement of resources and task executors to maximise their performance.

5.1. Static approaches using mini-batches

Spark Streaming [58] batches data in small time intervals (micro-batches) and passes it through deterministic parallel operations. Jobs are applied on DStreams according to users' application's action operations. Each job is portioned into tasks (the smallest execution unit in Spark) and a Spark's scheduler is responsible for their assignment to available resources. Scheduling micro-batch jobs in Spark Streaming to maximise performance and resource efficiency is quite cumbersome as a batch often involves multiple jobs with complex and dynamic dependencies in a workload.

By default, Spark's scheduler runs jobs in FIFO fashion and each application tries to use all available nodes. Although job dependencies can be captured with FIFO, this approach can result in increased latency when a long-running job delays jobs behind it. Users can limit the number of nodes that an application may use and set manually the level of parallelism but this will remain the same during the entire application execution. Low parallelism can cause inefficient resource utilisation while high parallelism can schedule more inter-dependent jobs and finally lead to poor performance. This kind of scheduling is static as the overall scheduling decisions are determined before the system runs. Zaharia et al. [58,65] compared Spark Streaming with Storm reporting their throughput. Storm was negatively affected by smaller record sizes and completed 115 K records/s/node for 100 byte records compared to 670 K of Spark's. Benchmarks between Spark Streaming and other DSMSs can be found in Section 4.3. A good scheduler, though, has to consider the dynamic resource demand between jobs, and other factors in scheduling, like the batch size, or the level of parallelism to be able to guarantee real-time restrictions in Spark Streaming. Default sequential job scheduling in Spark is inefficient.

5.2. Static approaches in operator-based systems

Resource allocation and task scheduling in *Storm* involves real-time decision making on how to replicate bolts and spread them across the nodes of a cluster. Storm has much lower latency than Spark

Streaming due to Spark's micro-batch execution model (even 50 time less based on Lu's et al. benchmark [27]). The user specifies the number of concurrent tasks to run for each component. Each of these tasks that run in parallel contains the same processing logic but work on different data and may be executed at different physical locations. Thus, tasks from a single bolt or spout are likely placed on different physical machines. The default scheduler uses a round-robin strategy to assign tasks to node's slots equally. Such a strategy is not optimal.

Aniello et al. [5] proposed two schedulers for Storm an offline and an online (see Section 5.4). The first one analyses the topology graph and identifies possible sets of bolts to be scheduled on the same node by looking at the way they are connected. Communication patterns among executors are examined, to place the most communicating executors to the same slot. Finally, the slots are assigned to worker nodes in a round-robin fashion. They tested their approach performance both on a general topology and in a more realistic setting. The latency of processing an event was improved by the offline scheduler with respect to the default one, around 20% and the inter-node traffic 13%.

Storm's default scheduling algorithms disregard resource demands and availability and usually become inefficient. It provides a scheduler (IsolationScheduler) that lets the user specify topologies to run on a dedicated set of machines within the cluster but the isolation setting is static [30]. Peng et al. [6] implemented *R-Storm*, a topology and resource-aware scheduling approach within Storm. Given the Storm topology, breadth first traversal (BFS) is used to create a partial ordering of components to place adjacent components in close succession to each other and schedule their tasks as close as possible. Having an ordered list of tasks to be scheduled, the candidate node is determined by finding the closest Euclidian distance in 3-d space between two points: task's requirements and particular node's available capacities. Scheduling computed by R-Storm provided 30–50% higher throughput than that computed by Storm's default scheduler. By co-locating tasks that communicate with each other on the same machine or same server rack, minimises network communication latency. R-Storm's CPU utilisation was 70–350% higher compared to Storm as R-Storm used fewer machines to produce to the same level of performance.

While R-Storm yields better performance than the default round-robin, it cannot control the performance when CPU sharing occurs. Smirnov et al. [7] proposed a performance-aware strategy that is independent of resources utilisation strategies employed by R-Storm and is based on a *genetic algorithm (GA)*. The main part of a genetic algorithm is its fitness function, whose aim is to build the schedule and evaluate its performance. Performance models of executors on specified nodes are needed to estimate their throughput and are built either on statistical data which is gathered during a run or on historic data. Each executor's performance depends not only on the node to which it is assigned but also on its parent executor. In their experiments, they allowed CPU-sharing between tasks and they proved that maximum tasks' performance can be achieved via minimum CPU sharing, consuming though the maximum number of cores. Their benchmarks results showed up to 40% better topology performance in terms of throughput of GA in comparison to R-Storm in heterogeneous clusters and 16% in homogeneous. GA scheduler was able to handle the high workload in several topologies, whereas R-Storm's and Storm's performance remained almost equal and low.

Eskandari et al. [66] presented a hierarchical adaptive scheduling scheme called *P-Scheduler*. The P-Scheduler is also a topology-aware strategy that uses weights on the graph edges, which store the data transfer rates. These rate values are obtained by initially submitting the known topology to the cluster and running it on the default Storm scheduler. The number of nodes needed to run a topology is computed on the basis of the topology load. These nodes share the entire topology and their communications are determined. The algorithm assigns the highest communication pairs of the topology to the same worker (JVM) and the other workers are loaded with a subdivision of the entire topology. The topology is partitioned by METIS, a specialised software for graph partitioning. By this arrangement, the inter-process and inter-worker traffic is reduced. The *P*—scheduler is also a workload-aware strategy, in the sense that most of the Storm topologies have almost similar workload and generally, no load balancing issue exists. The authors implemented P-Scheduler in a homogeneous cluster with 8 worker nodes and tested it on three topologies. The results have shown that P-scheduler reduces the

average tuple processing time be almost 50% as a result of reducing the inter-node and inter-worker communications.

Rychly et al. [64] proposed a scheduling algorithm on heterogeneous clusters that employs design-time knowledge and benchmarking techniques. To deploy processors to available slots while maintaining workload in a heterogeneous cluster, the system has to:

(a) get knowledge of nodes' performance characteristics for different types of computations and
(b) find a way to analyse the computation characteristics of incoming tasks and input data.

Detailed specification of a topology's individual nodes is provided at the cluster design-time by the user. As far as tasks' characteristics are concerned, benchmarking of every application on a particular cluster prior to its run in production is suggested to evaluate tasks' consumption profiles. To evaluate their approach, their prototype was implemented as a pluggable scheduler for Apache Storm. The performance gain of their scheduler in terms of all tuples processed by the whole application was 4.1% over Storm's standard scheduler and 17.5% over the worst possible scheduler. Applications employing GPUs and FPGAs for some of their components can benefit the most from their approach.

5.3. Dynamic approaches using mini-batches

As we already mentioned in Section 5.1, the default job scheduling in Spark Streaming is not sufficient. There is an option available when Spark is used with Mesos [67], which offers dynamic sharing of CPU cores. In this way, an application may give resources back to the cluster if it does not need them and request them again later when there is demand. This feature is useful when multiple applications share resources in the cluster but it may take a while for an application to gain back cores when it has work to do and cause less predictable latency.

Flink keeps track of distributed tasks, decides when to schedule the next task (or set of tasks), and reacts to finished tasks or execution failures. Flink's JVM processes (TaskManagers) provide a number of execution threads in the cluster, spawned to run one pipeline of parallel tasks. A pipeline consists of multiple successive tasks. The parallelism of Flink applications is determined by the degree of parallelism of streams and individual operators. Flink employs a schedule-once, long-running allocation of tasks. It uses an immediate scheduling and a queued scheduling algorithm. The first one returns a slot immediately when there is a request while the second one queues the request and returns the slot whenever it is available. Thus, in a sense, Flink's scheduler works in an arbitrary fashion. Nevertheless, the system is flexible to reconfigure pipelines to more or less workers and re-allocate application state on-demand. This approach minimises management overhead and allows adaptation to hardware or software changes or partial failures that can potentially occur [68,69]. Benchmarks between Flink and other stream processing systems can be found in Section 4.3.

Cheng et al. [8] proposed *A-scheduler*, an adaptive scheduling approach that dynamically schedules multiple jobs concurrently using different policies based on their dependencies. The data dependency between jobs is identified by profiling the DAG of an application while accepting a job submission (topology aware). A resource allocator applies Fair Scheduling for independent jobs and FIFO for dependent ones. It also collects performance statistics like end-to-end latency for each job and system throughput to automatically adjust the job parallelism settings and resource sharing policies (done by the Adaptive Tuning Optimiser). The tuning problem was formulated as a reinforcement learning process that uses a performance-aware approach. Their experimental results show that A-scheduler can reduce end-to-end latency by 42%, and improve workload throughput by 21% and energy efficiency by 13% compared to Spark Streaming.

Drizzle [70] is also a topology-aware strategy that uses the communication structure of a DAG. The main goal of this strategy is to decouple processing intervals from coordination intervals used for fault tolerance. To achieve this, the strategy uses a central scheduler that implements micro-batch groups scheduling, all groups at once. This avoids central scheduling bottlenecks and data processing

is completely decoupled from scheduling decisions. An adaptive group-size tuning algorithm inspired by TCP congestion control is used. During the execution of a group, counters are used to track the amount of time spent in various parts of the system and a policy analogous to AIMD determines the coordination frequency for a job. The tuples are processed within a range of milliseconds while coordinating functions take place within a range of seconds. Add-on strategies are used to improve performance. Specifically, pre-scheduling is used to let the worker machines track the data dependencies and run the task when dependencies are met. Query optimisation techniques are used to achieve better throughput. The strategy was compared to Spark and Flink in terms of end-to-end processing delay and failure recovery time and it was reported to have 3–4 times lower latency than Spark. Another comparison metric was failure recovery, which is performed 4 times faster compared to Flink.

A pre-scheduling framework is also proposed by Chen et al. called *Lever* [71]. The authors focus on the straggler problem; re-scheduling stragglers during the task execution period increases the processing time of the micro-batches and causes expensive data relocation, as the data have already been dispatched. Lever monitors and periodically collects and analyses the historical job profiles of the recurring micro-batch jobs and predicts stragglers. It then evaluates node capacity and chooses suitable helpers. It also makes re-scheduling decisions before the batching module dispatches the data to avoid increasing the processing time of the micro batch. Timely scheduling decisions minimise the processing latency. Chen et al. implemented Lever in Spark Streaming and when it was evaluated, it reduced job completion time by 30% to 40% and outperformed traditional techniques significantly.

When it comes to micro-batch based systems, the matter of batch-sizing is a factor of crucial importance as it affects latency and can facilitate the scheduling and rescheduling of tasks and data. *Das et al.* [72]' online algorithm automatically adapts the batch size to affect the system's performance. By dynamically adjusting the batch interval, fluctuations within operating conditions can be handled and the total delay of every event can be controlled. Authors used a control module to gather information from past job statistics (e.g. processing time, queuing delay, etc.) to learn the characteristics of the workload and adapt online the batch size when changes in data rates, workload and resources occur. Their control algorithm is based on the Fixed-point Iteration optimisation technique and their experiments indicate that system stability can be ensured for a wide range of workloads (filter, reduce, join, window), despite large variations in data rates and operating conditions. In the evaluation, their approach was able to ensure latency that was comparable to the minimum latency achieved with any statically defined batch interval method. Nevertheless, it is vulnerable to large loop delays. Sudden large changes in the workload can introduce large delays.

Liao et al. [73] also explored the effects of batch size on scheduling decisions and the performance of streaming workloads in Spark Streaming. Their idea relies on dynamically adjusting batch intervals, according to the number of events arrived in the current interval. Reducing the interval near data peaks proved to be a good choice to smooth the overall delay time. Based on the existing scheduling framework of Spark Streaming, a timer is used at the starting stage for the batch division. They used typical streaming benchmarks to demonstrate that their policy is effective in handling unstable streaming inputs leading to limitation of the total delay of events.

5.4. Dynamic approaches in operator-based systems

Storm's default scheduling algorithm is static and remains far from optimal. Generic scheduling solutions for provisioning to multiple applications competing for cloud resources like Mesos and YARN assume that an application already knows the amount of resources it needs, and how to distribute these resources internally. Fu et al. [63] designed and implemented *DRS*, a dynamic resource scheduler that applies to operator-based stream processing systems. Their algorithm takes into account the number of operators in an application and the maximum number of available processors that can be

allocated to them and tries to find an optimal assignment of processors that results in the minimum expected total sojourn time. They estimated the total sojourn time of an input by modelling the system as an open queuing network (OQN). The performance model is built based on a combination of one of Erlang's models and the Jackson network. The system monitors the actual total sojourn time and checks if the performance falls, or if the system can fulfil the constraint with less resources and reschedules if necessary. It repeatedly adds one processor to the operator with the maximum marginal benefit, until estimated total sojourn time is no larger than a real-time constraint parameter. DRS uses Storm's streaming processing logic and demonstrates robust performance, suggesting the best resource allocation configuration, even when the underlying conditions of the queuing theory that it uses are not fully satisfied. In general, DRS' overhead is less than milliseconds in most of the cases tested resulting in small impact on system's latency.

Apart from using all available worker nodes, Storm's scheduler does not consider links between tasks that are about to be hosted in these nodes. Inter-node and inter-process traffic are factors that make a significant impact on system's performance. *Meng-meng et al*. [74] proposed a dynamic task scheduling approach that considers links between tasks and reduces traffic between nodes by assigning tasks that communicate with each other to the same node or adjacent nodes. The topology is obtained by recording workload of nodes and communication traffic through switches a priori. They used a matrix model to describe the real-time task scheduling problem. Their processing procedure tries to reduce traffic between nodes through switches, cut off bandwidth pressure and balance workload of nodes by selecting the appropriate host node when a trigger (either node-driven or task-driven) occurs. They evaluated their algorithm by deploying their own stream processing platform and compared their solution with the algorithms that are built in Storm and S4 [75] using load balance and communication traffic through switches as indicators. As the number of jobs running in these platforms was growing the load balance in SpeedStream was better. Moreover, less stream data flowing through switches were detected and this traffic was reduced, relieving the bandwidth pressure of the cluster.

Aniello's et al. [5] second scheduler is also a traffic-aware scheduler that works dynamically and extends the topology-aware approach used in the first one. To do so, it monitors the effectiveness of the schedule at runtime using a performance log and adapts the allocation of executors to the evolution of the load and according to the communication patterns of the application to reduce the inter-node and inter-slot traffic in the cluster. Nevertheless, the scheduler does not check the whole communication pattern between executors and considers each pair in isolation. In this way, some communicating pairs might end up in different nodes. The custom scheduler can periodically check if a new more efficient schedule in terms of inter-node traffic can be deployed. The latency of processing an event was below 20–30% and the inter-node traffic below 20% with respect to the default Storm scheduler in both tested topologies (the general one and the one based on the realistic set).

T-Storm [76] is another attempt that tries to minimise inter-node and inter-process traffic. Workload and traffic load information are collected at runtime by load monitors to estimate future load using a machine learning prediction method. A schedule generator periodically reads the above information from the database, sorts the executors in a descending order of their traffic load, and assigns executors to slots. Executors from one topology are assigned in the same slot to reduce inter-process traffic. The total executors' workload should not exceed worker's capacity and the number of executors per slot is calculated with the help of a control parameter. T-Storm consolidates workers and worker nodes to achieve better performance with even less number of worker nodes, enables hot-swapping of scheduling algorithms and adjusts scheduling parameters on the fly. T-Storm's evaluation shows that it can achieve over 84% and 27% speed-up of average processing time on lightly and heavily loaded topologies respectively with 30% less number of worker nodes, compared to Storm.

System overload is also a matter of interest for Liu et al. [77]. They proposed a dynamic assignment scheduling (DAS) algorithm for big data stream processing in mobile Internet services. The authors generate a structure called stream query graph (SQG), based on the operators and the relations between the corresponding input and output. SQG is a direct acyclic graph and an edge between

two nodes represents a task queue. The edge weight is the number of tasks in the queue. The minimum-weight edge is selected to send tuples and a buffer list is set to store some tuples before next scheduling. DAS' scheduling strategy is updated continuously by every logic machine separately. By splitting the general scheduling problem into the common sub-problem of every operator, the overhead is reduced and accuracy is improved. The authors compared DAS to SAMR [78], a dynamic scheduling algorithm based on MapReduce. DAS was found to have has shorter response time and takes up less system resource, mainly because SAMR needs continuously decompose tasks and read local history information, thus more disk operations are involved.

Elasticity is a matter of crucial importance in online environments to determine how to scale for data stream fluctuating with time and schedule resources according to the current arrival rate of a stream. Dawei et al. [79] proposed an elastic online scheduling framework (*E-stream*), that works with multiple DAGs, using the available capacity of computing nodes and the input rate of the data stream. E-stream quantifies computation and communication cost, relationships between the input and output stream of a vertex, and adjusts the degree of parallelism of vertices in the graph that has to be scaled in our out. Subgraphs are further constructed to reduce the communication cost between related vertices and DAG's response time. Each subgraph is substituted by a logically equivalent vertex and is treated as a 'vertex' in the scheduling phase. When it comes to scheduling, E-stream monitors whether DAGs require more resources and reschedules them using a priority-based earliest finish time first (EFT) strategy by keeping the system fairness degree guaranteed. The output is a max-min fairness-based multiple DAGs scheduling sequence with makespan (total elapsed time required to execute a graph) guaranteed. E-stream was developed based on Storm. When compared to Storm, it yielded reasonable response time even when 50 DAGs were employed, which was shorter than that of the default Storm scheduler (about 3 and 7 times faster for different applications). Moreover, E-stream's fairness degree was better than that of Storm's scheduler.

Floratou et al. [9] focus on service level objectives (SLO) that have to be maintained in case of unpredictable load variation and hardware or software performance degradation. They introduced Dhalion, a system with self-regulating capabilities and implemented it on top of Heron. Dhalion periodically invokes a policy that examines the status of the streaming application collecting metrics from Heron's metrics managers, detects possible problems (e.g. slow processing, lack of resources, etc), diagnoses them, and perform appropriate actions. Two policies were designed. The first one provisions resources dynamically for throughput maximisation, taking into consideration the input data load (workload variations). The second one takes as input a throughput SLO, keeps tracking the observed throughput while the topology is running, and auto-tunes the Heron application (adjusting the parallelism of spouts or bolts) or provide necessary resources provision if needed. In their evaluation experiments, they tested both policies. The first policy adjusted the topology resources on-the-fly when workload spikes occurred and backpressure was observed. The second policy auto-tuned the topology successfully as it reached a steady state, when the throughput observed was equal or higher to the throughput SLO.

Unlike most approaches, *Cardellini et al.* [80] did not use a clustered environment but a distributed system, spread among multiple small data centres. They implemented a distributed algorithm that actually adopts a network-aware scheduling algorithm (proposed by Pietzuch et al. [81]) to Storm. The QoS attributes (latency, utilisation and availability) are considered known and they focus on modelling data processing between each pair of nodes (cost space) and on placing operators (operator based) in this cost space. The QoS awareness is provided by the QoSMonitor and Vivaldi's algorithm [82] provides accurate estimations between pairs of nodes. Finally, the AdaptiveScheduler acquires this information to identify which executors can be effectively relocated to finally place them in the appropriate node (the MAPE reference model [83]is used to support this procedure). Due to rearranging the operators to nodes with higher availability, latency is reduced about 72% compared to Storm and almost 17% less tuples had to be resent. Moreover, the proposed scheduler provides more efficient load balancing.

Safaei [84] presented a whole architecture for a real-time streaming engine in a multiprocessing environment, paying attention to the value of velocity. Queries and some of their characteristics are assigned to the clusters but each query is accepted for processing if its deadline can be satisfied. System's response time to a query is computed using a function (details about the function in [85]). This performance-aware algorithm compares the execution time of each query to the query's deadline. The algorithm selects the highest priority query and allocates it to the proper cluster of processors using a First-Fit algorithm based on a utilisation factor (it needs the execution time of each query to be computed). Each cluster selects from its waiting queue of queries using the EDF (Earliest Deadline First) algorithm. The selected query is then processed in parallel via a proposed deadline-aware dispatching method; a tuple is processed by a processor and then forwarded to the best next processor to continue its processing based on the job's deadline. Several parameters were checked to evaluate and compare this prototype to Storm. The results showed that this approach outperforms Storm in terms of proportional deadline miss ratio (a metric for the evaluation of real-time stream processing systems, $\sim$ 50% of Storm), tuple latency ($\sim$ 66% of Storm) and throughput ($\sim$ 1.4% of Storm) while had some penalties as far as it concerns memory usage ($\sim$ 1.2 of Storm) and tuple loss ($\sim$ 1.9 of Storm).

Table 2 presents a classification of the aforementioned approaches and the addressed issues.

5.5. Discussion

As we saw, in recent years, a lot of researchers presented different task scheduling approaches and algorithms. The diversity of the clusters where the evaluation of the aforementioned systems took place does not let us make safe comparisons between them. Moreover, differences in applications' inherent characteristics and in data streams used (e.g. their transfer rate, their realistic nature etc.) are also major impediments to safe comparisons. Our taxonomy reveals that there are more available enhancements that use the operator based model in stream processing than the micro-batch. However, there are enough scheduling approaches for batch processing systems like Spark that might also fit in systems using the micro-batch model but they are not included in our review, since they have not been tested in stream processing. It also seems that most of the effective scheduling decisions in stream processing are made online or at least are based on prior online knowledge. The observed parameters that affect decisions of the presented scheduling techniques are mainly topology, available resources and network, workload, and system's performance.

Topological issues: Based on our review, when it comes to static approaches most scheduling decisions in the systems studied rely on the topology that has to be run. Topology plays also the second most important role in decision making in dynamic approaches (it comes second after workload characteristics). When there is not a large processing burden, the system throughput relies heavily on the network communication latencies. Given the topology, communication patterns can be found, inter-node and inter-process traffic can be reduced, and the throughput is then expected to increase dramatically. For example, the linear and star topologies (tested in R-Storm) necessarily involve larger number of communications and it is no surprise that the throughput improvements are more significant compared to Storm, when these topologies were tested, since Storm does not take into account the inter-node and inter-process traffic. Assigning the most communicating tasks together on one node or rack is a need that differentiates stream processing from batch processing, which pays attention to data locality instead (assigning computations to the nodes where the required data is stored).

Resources issues: The processing of big data requires a large amount of CPU cycles, memory, network bandwidth, and disk I/O. Especially in stream processing, memory becomes of crucial importance. It is essential to effectively schedule the tasks, in a manner that minimises task completion time and increases utilisation of resources. Resource-awareness is a common need both in the static approaches and dynamic approaches studied. Such strategies try to take advantage of node utilisation. In GA Storm, the authors have shown that their strategy performs better when the tasks are shared between

Table 2. An overview of scheduling approaches in stream processing big data frameworks.

System	Execution model	Scheduling decisions	Awareness	Tools	Comparison with	Evaluation metrics
Spark Streaming	Micro-batches	Offline	–	FIFO	Storm	**Better**: Throughput: 5×–6×
Storm	Operator based	Offline	– –	Round Robin	Spark Streaming –	**Better**: Latency: 50× –
Aniello et al. (static)	Operator based	Offline	Topology	Round robin	Storm	**Better**: Latency: 20%, **Less**: Inter-node traffic: 13%
R-Storm	Operator based	Offline	Topology, Resources	BFS	Storm	**Better**: Throughput: 30–50% **Less:** CPU Utilisation: 70–350%
GA Storm	Operator based	Offline	Topology Performance	Genetic algorithm, Monitoring system	Storm R-Storm	**Better**: Throughput: 16–40% (R-Storm)
P-Scheduler	Operator based	Offline	Topology, Traffic, Workload	METIS, Monitoring log	Storm	**Better**: Throughput: 50%
Rychly et al.	Operator based	Offline	Resources, Performance	Round robin, Benchmarking (with monitoring component)	Worst scheduler, Storm's Even Scheduler	**Better**: Tuples processed by app: 4.1% (over Storm)-17%(Worst scheduler)
Flink	Operator based(Hybrid)	Online	Resources		Spark Streaming, Storm	**Better**: Throughput, **Less**: Memory utilisation
A-Scheduler	Micro-batches	Online	Topology, Performance	Reinforcement Learning Process, FIFO, FAIR	Spark Streaming	**Better**: Latency: 42%, Throughput: 21%, Energy efficiency: 13%
Drizzle	Micro-batches	Online	Topology, Performance	TCP congestion control, AIMD policy queues	Spark, Flink	**Lower**: Processing latency: 3.5 × (Spark), similar(Flink), Adaptability to failures: 4× faster than Flink
Lever	Micro-batches	Online	Resources	monitors	Spark Streaming	Job completion time: 30–40% less
Das et al.	Micro-batches	Online	Resources Workload	Control module, Fixed-point Iteration Optimisation Technique	Statically defined batch intervals methods	Stability, Latency: equal to minimum value
Liao et al.	Micro-batches	Online	Workload	Timer	Spark Streaming	**Better**: Latency
DRS	Operator based	Online	Resources, Performance	Erlang queuing model, Jackson network, Greedy	Storm	Rebalancing for optimal solution, **Low**: Computational overhead

(*continued*)

Table 2. Continued.

System	Execution model	Scheduling decisions	Awareness	Tools	Comparison with	Evaluation metrics
Meng-meng et al.	Operator based	Online	Topology Workload Traffic	Matrix model, Monitor	Storm S4	**Better**: Load balance **Less:** Communication traffic
Aniello et al. (dynamic)	Operator based	Online	Workload Traffic	Performance log	Storm	**Better**: Latency: 20–30%, **Less:** Inter-node traffic: 20%
T-Storm	Operator based	Online	Workload, Traffic	Load monitors, Machine learning prediction method	Storm	**Better**: Throughput: 27%, 84% **Less**: Number of worker nodes: 30%
DAS	Operator based	Online	Topology, Workload	Queues, Buffer list	SAMR (MapReduce)	**Less**: Number of resources, **Lower**: Response time
E-stream	Operator based	Online	Data stream rate, Topology Resources	Earliest finish time first strategy, Max–min fairness-based strategy	Storm	**Better**: Response time: 3–7× **Better**: Fairness degree
Dhalion	Operator based	Online	Workload SLO	Heron metrics managers		
Cardellini et al.	Operator based	Online	Network, QoS	Pietzuch scheduling algorithm, Vivaldi algorithm, MAPE reference model	Storm	**Better**: Latency: 72%, **Less**: Tuples resent: 17.7%, **Better**: Load balance
Safaei	Operator based	Online	Performance	System's response time function, First-Fit, EDF	Storm	**Less**: PDMR: 50%, **Better**Latency: 66%, Throughput: 1.4%, **More**: Memory usage: 1.2 ×, Tuple loss: 1.9 ×

the maximum amount of cores, thus, we have the fewer possible number of tasks per core. Maximum tasks' performance can be achieved via minimum CPU sharing, consuming though the maximum number of cores. This is an obvious assumption, however, it poses a question: How will a resource-aware strategy fully utilise a CPU? As an answer to this question, most strategies presented above (except GA Storm) have to fully assign one CPU per component of the topology. This means that, in a heterogeneous environment, the CPUs with higher computational power should be scheduled to process tasks that require heavier processing, while in a homogeneous environment, scheduling is based on the assumption that the CPUs available have enough capacity.

Cluster heterogeneity affects the static scheduling decisions. In the aforementioned systems, authors usually indicate if the proposed schemes target at a homogeneous or at a heterogeneous environment. In R-Storm, the nodes on which the tasks are scheduled are determined by a distance function, that is based on the resource availability. However, the scheduling strategy was implemented for homogeneous clusters, where all the CPUs are assumed to have similar computational power. In cases where different CPU architectures exist, resource-aware scheduling cannot be easily applied. For the example of R-Storm, a CPU selected from a group of available CPUs based on the 'nearby' available resources criterion, is not guaranteed to work as expected (complete processing), unless it is known in advance that all the CPUs are identical. Smirnov et al. proved experimentally that throughput is highly determined by the type of CPU. Aniello et al. have dealt with heterogeneous nodes and considered the CPU speed in their predictions. Their strategy was based on moving tuples across a chain of hops after they have been sent by a spout, until its processing ends up in an ack bolt. As an example of moving tuples, if an executor is taking 10% CPU utilisation on a 1GHz CPU and migrates on a node with 2GHz CPU, the CPU utilisation would become 5%. With similar CPUs, such a scheduling would require load balancing to increase utilisation. However, load balancing necessarily involves careful selection of the tuples assigned to each CPU and some a priori knowledge regarding the size of the tuples. Moreover, Rychly et al. have shown that their resource-aware, worst, standard, and best scheduling approach provide the same results on a homogeneous platform while significant improvement of the best scheduling over the worst and the average scheduling is shown with increasing heterogeneity, because the best schedule fully utilises the differences in hardware. Another topology-aware strategy that could operate on a heterogeneous platform is the P-Scheduler. In P-Scheduler, the authors use a homogeneous platform. However, in this case, their main goal is to minimise the network latencies by assigning highly communicating tasks to the same nodes. In such a scheme, the CPU architecture is not of the main importance, however, the authors suggest some type of heterogeneity to the transfer rates of the available streams, so that they can use faster streams to transfer tuples with higher speed when it is necessary. Although they try to reduce the transfer time, they claim that this will also reduce the processing time of the tuples.

Workload issues: When it comes to dynamic approaches, workload seems to influence most scheduling decisions. Workload characteristics become of crucial importance when it comes to micro-batch processing systems as they help to determine the appropriate batch size on scheduling decisions (Liao' et al. and Das' et al. systems). Generally, the idea to deal with this issue is to try to adjust the stream sizes accordingly. When smaller streams are used, there is a faster adaptation to system changes. In operator-based systems, workload-aware approaches can help towards mitigating overloading in a worker node. When a machine's computational resources are not adequate to handle the processing needs, its capacity can be set to a fraction of its actual capacity to prevent overloading (just like T-Storm does) but this is not enough. Large load spikes lead to bottlenecks, possible backpressures and the overall system throughput decreases. Moreover, frequent load balancing and state

migration techniques can increase overhead and consequently latency. Nevertheless, the reduced rate of scheduling can lead to inaccuracy. Workload fluctuations demand elaborate handling.

Elasticity refers to the ability of a cloud to allow a service to allocate additional resources or release resources on demand to match the application workload. Nevertheless, without adjusting the parallelism of components, a topology's throughput reaches a ceiling above which adding more machines will not improve performance. Scheduling a topology among unnecessary number of machines can cause an increase in communication latency. Much effort and several configurations are usually needed by users to determine the degree of parallelism at each stage of a streaming pipeline as users have limited knowledge about the runtime behaviour of the system. Systems like E-Stream monitor the rate of stream and in case it increases, they create some new instances for a number of vertices. Self-regulating streaming systems like Dhalion also help towards this direction, as they take into account the specifications of streaming applications like the input data rate, as well as policies defining the users' objective to heal the problem dynamically. Another interesting approach refers to splitting the general scheduling problem into sub-problems of each operator (just like DAS does), to reduce overhead and improve accuracy.

Two more stream-oriented approaches that could be proposed are: (1) Use of priorities: When there is some mechanism to assign priorities to certain streams, then we keep on processing these streams (thus their processing is not deferred) and we move less important streams to the newly incoming nodes for later processing, and (2) Dropping less important streams: In such a solution, the less important tuples can be dropped to reduce latencies. The challenge here is to decide how many tuples to drop in order to maintain accuracy. This problem can be reduced to an optimisation problem of what is the optimal number of tuples to be dropped in order to maintain a certain level of accuracy.

Elasticity becomes even more challenging in stream processing environments where computations are generally stateful. Guaranteeing fault-tolerance and dynamic load balancing for stateful operators demands state transfer which is quite cumbersome. As we saw, Flink uses special mechanisms for state handling (the interested reader can refer to Hoffman et al. [86] and Del Monte et al. [87] for more details on state migration).

Performance issues: A system's performance can be either monitored during runtime or derived from old executions' or benchmarks' statistics. Monitoring performance characteristics like total sojourn time (DRS), query execution time (Safaei's approach) or throughput (A-Scheduler), can help systems either adjust their behaviour accordingly online or construct models for effective decision making. The GA Storm scheduler is also an example of performance-aware strategy, but its scheduling is static. CPU sharing is used to maximise performance but for performance-aware strategies, there is a high dependence on the environment's homogeneity. In homogeneous environments, the GA Storm strategy fails to improve performance, because the estimated throughput of all the executors seems to be similar while evolving executors are generated from their parents.

Monitoring of a system's performance becomes of critical importance in the presence of faults. Different *fault tolerance* techniques are important for the efficient utilisation of provided resources. Streams should be able to recover from failures to keep streaming applications running with no problems. In stream processing systems, fault tolerance is a wide area of research and it is attracting more and more attention. In this paragraph we only make some useful remarks with regard to fault recovery of the DSMSs under study, using appropriate scheduling techniques. Especially, when it comes to dynamic strategies, where decisions are taken during runtime, the failure recovery strategies are of primary importance. As shown in the literature, the choice of a strategy is greatly affected by the resources and the time required for recovery. Generally, the scheduling strategies described above handle fault recovery in one of the following ways:

(a) *Checkpoint-based recovery*. In this strategy, a vertex periodically checkpoints its stream. When the vertex fails, it will load the most recent checkpoint and resume execution. Checkpointing introduces overheads in normal execution and it is not an ideal solution for nodes with high load. This is due to

the fact that the existence of many checkpoints necessarily requires large data structures, thus increasing its total cost. One way to reduce the cost is to place checkpoints only at the end of the streams. However, when the stream is large, too many records need to be reproduced in case of failure. Another option is to handle failures in pre-scheduling, but there is no guarantee that such a solution is optimal in a real-time system.

(b) *Replication-based recovery*. Another strategy is to have multiple instances of the same node run at the same time: they can be connected to the same input streams and output streams. When one instance fails, recovery is achieved by getting the same snapshot from another instance. Apparently, there is no need to use large databases in this case, but there are three drawbacks: (a) the existence of multiple instances, (b) the choice of the proper nodes that run these instances, which sometimes requires careful selection depending on issues like distance, and (c) difficulty in predicting the streams that would have to be replicated beforehand, thus scheduling of replicas is another issue that needs to be considered. In such cases, efficient approaches should be developed for calculating the minimum replication, so that the reliability requirement is met.

As we see, the performance of a DSMS depends on multiple factors. Stream scheduling seems to be more efficient when it is online, including continuous, incremental scheduling decisions that account for changes in the streams' arrival rate and cluster resource utilisations. The capacity and the capabilities of the underlying cluster environment in which processing is taking place will always limit its performance. It's certainly clear that task scheduling constitutes a critical factor for systems' performance. Achieving low-latency, high-throughput processing of streams when resources may be scaled up or down, requires an effective task scheduling that reduces the number of task migrations, allocates the number of dependent and independent tasks in a near optimal manner to decrease the overall computation time of a job, and improves the utilisation of cluster resources.

6. Conclusions

Data that is received in real-time can become way too valuable at the time it arrives and supports valuable decision making. Several cloud-based systems emerged to enable distributed processing of streams of big data. Mechanisms used by prominent DSMSs to face the challenges posed by stream processing in the context of big data were presented in this work. We also described a taxonomy that could facilitate the comparison of different features offered by stream processing frameworks. Based on this taxonomy, we provided an overview of four open-source stream processing frameworks. Our study provides an insight of factors that have to be considered when selecting a platform, given a specific use case. For example, Flink is a good choice if complex stream processing is needed. However, Spark Streaming is a more mature project and has a bigger community. Storm is also a mature project and can provide better latency with fewer restrictions, but cannot guarantee state consistency. It also offers just basic building blocks for composing a topology to users, whereas Spark and Flink expose a higher level API as declarative systems. Near real-time streaming, systems compete against each other on several factors but there is not a clear winner yet. Each of the platforms mentioned here has their advantages and disadvantages. Understanding strengths and limitations of the aforementioned frameworks in tandem with users' needs, makes it easier to pick an appropriate solution. Complexity of stream computing and diversity of workloads expose challenges to benchmark these systems, but small changes in configuration's parameters or in the available infrastructure can change the results. Consequently, further research is needed to test and evaluate distributed stream processing platforms in different cloud experiments using not only criteria referring to performance but also to security or integration with other tools. Large and active communities, that support these processing projects, continue to innovate but could benefit from further suggestions and improvements on these tools.

The performance of a stream processing system depends on multiple factors. Task scheduling has evolved to a critical factor that can significantly affect the performance of cloud frameworks. The mechanisms used by the aforementioned systems are not very sophisticated and do not consider the underlying available infrastructure which can be heterogeneous in some cases. Thus, task scheduling

has become a matter of interest for a lot of researchers. However, to the best of our knowledge, there is no extensive study on task scheduling approaches for big data stream processing frameworks that classifies and discusses the presented approaches. We analysed 22 different approaches and identified the most important factors affecting the decisions of the proposed schedulers, namely topology, resources, system's performance, and workload. We showed how decisions based on these factors affect the systems under consideration. Our survey allowed us to classify the scheduling issues in different categories. Most static approaches aim at providing an initial placement strategy to ensure minimal labouring in the cluster. However, the changes in workload and system conditions enforce the need for runtime scheduling techniques that help in reducing the imbalances in resource utilisation and improves the performance of the stream processing system.

Our survey reveals some limitations, and we can outline some possible future directions. The current taxonomy does not include commercially available stream processing platforms and relies only on open-source stream processing platforms with documentation and source code freely available. There are also more promising frameworks like Apache Apex, a Hadoop YARN native platform that unifies stream and batch processing. Lately, there is an initiative to bring different batch and stream processing engines on a common ground. For example, Apache Beam can be used to write APIs independent of the framework, and then can run the same code on any engine. Such initiatives and their mechanisms could be considered for future research. Moreover, most of the approaches presented are evaluated based on metrics like latency, throughput and fault-tolerance but as far as fault-tolerance is concerned extensive research is still conducted. There are only a few papers that calculate just the number of tuples lost. A risk and cost analysis of stream processing platforms using different scheduling techniques could also be a matter of interest in the future, as the cost has an important role in service-level agreements (SLA) between service providers and clients.

Disclosure statement

No potential conflict of interest was reported by the authors.

ORCID


Nicoleta Tantalaki http://orcid.org/0000-0003-3839-4829

References

[1] Chen M, Mao S, Liu Y. Big data: a survey. Mobile Netw Appl. 2014;19(2):171–209.
[2] Hashem , Yaqoob , Anuar , Mokhtar S, Gani A, Khan S. The rise of "big data" on cloud computing: review and open research issues. Inf Syst. 2015;47:98–115.
[3] Talia D. Clouds for scalable big data analytics. ACM Comput. 2013;46(5):98–101.
[4] Stonebraker M, Cetintemel U, Zdonik U. The 8 requirements of real-time stream processing. ACM SIGMOD. 2005;34(4):42–47.
[5] Aniello L, Baldoni R, Querzoni L. Adaptive online scheduling in storm. Proceedings of the 7th ACM International Conference on Distributed Event-based Systems; Arlington, TX. 2013. p. 207–218.
[6] Peng B, Hosseini M, Hong Z, et al. R-Storm: resource-aware scheduling in storm. Proceedings of the 16th ACM Annual Middleware Conference; Vancouver, Canada. 2015. p. 202–215.
[7] Smirnov P, Melnik M, Nasonov D. Performance-aware scheduling of streaming applications using genetic algorithm. Procedia Comput Sci. 2017;108:2240–2249.
[8] Cheng D, Chen Y, Zhou X, et al. Adaptive scheduling of parallel jobs in spark streaming. Proceedings of the IEEE INFOCOM 2017 – IEEE Conference on Computer Communications; Atlanta, GA. 2017. p. 1–9.
[9] Floratou A, Agrawal A, Graham B, et al. Dhalion: self-regulating stream processing in heron. Proceedings of the VLDB Endowment, Vol. 10(12); Munich, Germany. 2017. p. 1825–1836.
[10] Liu X, Iftikhar N, Xie X. Survey of real-time processing systems for big data. Proceedings of the 18th International Database Engineering & Applications Symposium (IDEAS), Porto, Portugal. ACM; 2014. p. 356–361.
[11] Ranjan R. Streaming big data processing in datacenter clouds. IEEE Cloud Comput. 2014;1(1):78–83.
[12] Hesse G, Lorenz M. Conceptual survey on data stream processing systems. Proceedings of the IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS); Melbourne, VI. 2015; p. 797–802.

[13] Georgiadis C. An evaluation and performance comparison of different approaches for data stream processing [dissertation]; 2016. Available from: http://urn.kb.se/resolve?urn = urn:nbn:se:uu:diva-307214
[14] Assuncao MD, Veith AD, Buyya R. Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. J Netw Comput Appl. 2018;103:1–17.
[15] Kamburugamuve S, Fox G. Survey of distributed stream processing. 2016; 10.13140/RG.2.1.3856.2968.
[16] Singh M, Hoque M, Tarkoma S. A survey of systems for massive stream analytics. 2016; arXiv:1605.09021v2.
[17] Singh D, Reddy C. A survey on platforms for big data analytics. J Big Data. 2014;2(1):8.
[18] The Apache Software Foundation. Welcome to Apache Hadoop [Internet]. 2014. Available from: http://hadoop.apache.org/
[19] The Apache Software Foundation. Apache Spark [Internet]. 2018. Available from: http://spark.apache.org/
[20] Cugola G, Margara A. Processing flows of information: from data stream to complex event processing. ACM Comput Surv. 2012;44(3):15.
[21] Shahrivari S. Beyond batch processing: towards real-time and streaming big data. Computers. 2014;3(4):117–129.
[22] Namiot D. On big data stream processing. Int J Open Inform Technol. 2015;3(8):48–51.
[23] Apache Software Foundation. Introduction to Apache Flink [Internet]. 2016. Available from: https://flink.apache.org/introduction.html
[24] Cordova P. Analysis of real time stream processing systems considering latency. Data Science Association [Internet]. 2015. Available from: http://www.datascienceassn.org/content/analysis-real-time-stream-processing-systems-considering-latency
[25] Chintapalli S, Dagit D, Evans B, et al. Benchmarking streaming computation engines: storm, flink and spark streaming. Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW); Chicago, IL. 2016. p. 1789–1792.
[26] Perera S, Perera A, Hakimzadeh K. Reproducible experiments for comparing apache flink and apache spark on public clouds. 2016; arXiv:1610.04493.
[27] Lu R, Wu G, Xie B, et al. StreamBench: towards benchmarking modern distributed stream computing frameworks. Proceedings of the IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC); London, UK. 2014. p. 69–78.
[28] Karimov J, Rabl T, Katsifodimos A, et al. Benchmarking distributed stream data processing systems. Proceedings of the IEEE 34th International Conference on Data Engineering (ICDE); Paris, France. 2018. p. 1507–1518.
[29] The Apache Software Foundation. Spark Streaming-Apache Spark [Internet]. 2017. Available from: http://spark.apache.org/streaming/
[30] The Apache Software Foundation. Apache Storm [Internet]. 2017. Available from: http://storm.apache.org/
[31] The Apache Software Foundation. Apache Heron-A realtime, distributed, fault-tolerant stream processing engine from Twitter [Internet]. 2017. Available from: https://apache.github.io/incubator-heron/
[32] The Apache Software Foundation. Samza-What is Samza? [Internet]. 2016. Available from: http://samza.apache.org
[33] The Apache Software Foundation. Apache Kafka-A distributed streaming platform [Internet]. 2017. Available from: https://kafka.apache.org/
[34] Li R, Hu H, Li H, et al. MapReduce parallel programming model: a state-of-the-art survey. Int J Parallel Program. 2014;44(4):832–866.
[35] Rajeshwari U, Babu BS. Real-time credit card fraud detection using streaming analytics. Proceedings of the 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT); Bangalore, India. 2016. p. 439–444.
[36] Peng C-Z, Jiang Z-J, Cai X-B, et al. Real time analytics processing with MapReduce. Proceedings of the International Conference on Machine Learning and Cybernetics, Vol. 4; Xian, China. 2012. p. 1308–1311.
[37] Carbone P, Gevay GE, Hermann G, et al. Large-scale data stream processing systems. In: Zomaya A, Sakr S, editors. Handbook of big data technologies. Cham: Springer; 2017. p. 219–260.
[38] Bhartia R. Amazon kinesis and apache storm. Building a real-time sliding-window dashboard over streaming data. Amazon Web Services [Internet]. 2014. Available from: https://d0.awsstatic.com/whitepapers/building-sliding-window-analysis-of-clickstream-data-kinesis.pdf
[39] IBM. InfoSphere Streams [Internet]. 2016. Available from: http://www-01.ibm.com/software/data/infosphere/streams/
[40] Philip Chen CL, Zhang CY. Data-intensive applications, challenges, techniques and technologies: a survey on big data. Inf Sci (NY). 2014;275:314–347.
[41] SQLStream [Internet]. 2018. Available from: https://sqlstream.com
[42] Begoli E, Camacho-Rodríguez J, Hyde J, et al. Apache calcite: a foundational framework for optimized query processing over heterogeneous data sources. Proceedings of the International Conference on Management of Data (SIGMOD '18); New York, NY. 2018. p. 221–230.
[43] Owens T. Survey of event processing [Internet]. 2007. Available from: http://www.dtic.mil/dtic/tr/fulltext/u2/a475386.pdf
[44] Patroumpas K, Sellis T. Maintaining consistent results of continuous queries under diverse window specifications. Inf Syst. 2011;36(1):42–61.

[45] Akidau T, Bradshaw R, Chambers C, 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; Kohala Coast, Hawaii. 2015. p. 1792–1803.
[46] Affetti L, Tommasini R, Margara A. Defining the execution semantics of stream processing engines. J Big Data. 2017;4:12.
[47] Yang H. Solving problems of imperfect data streams by incremental decision trees. J Emerg Technol Web Intell. 2013;5(3):322–331.
[48] Gudivada V, Apon A, Ding J. Data quality considerations for big data and machine learning: going beyond data cleaning and transformations. Int J Adv Softw. 2017;10(1):1–20.
[49] Hwang J-H, Balazinska M, Rasin A, et al. High-availability algorithms for distributed stream processing. Proceedings of the 21st International Conference on Data Engineering (ICDE); Tokoyo, Japan. 2005; p. 779–790.
[50] Zaharia M. An architecture for fast and general data processing on large clusters. New York (NY): Association for computing machinery and Morgan & Claypool; 2016. (ACM book series).
[51] The Apache Software Foundation. Welcome to Apache ZooKeeper [Internet]. 2014. Available from: http://zookeeper.apache.org
[52] Rodriguez-Mazahua L, Rodriguez-Enriquez C-A, Sanchez-Cervante J-L, et al. A general perspective of big data: applications, tools, challenges and trends. J Supercomput. 2016;72(8):3073–3113.
[53] Karau H, Konwinski A, Wendell P, et al. Learning-spark. Sebastopol (CA): O'Reilly Media; 2015.
[54] Friedman E, Tzoumas K. Introduction to Apache Flink. O'Reilly Media [Internet]. 2016. Available from: https://mapr.com/ebooks/intro-to-apache-flink/chapter-5-stateful-computation.html
[55] Simonassi D, Eisbruch G, Leibiusky J. Getting started with storm. Sebastopol (CA): O'Reilly Media; 2012.
[56] Nasir M. Fault tolerance for stream processing engines. 2016; arXiv:1605.00928v2.
[57] Hagedorn S, Gitze P, Saleh O, et al. Stream processing platforms for analyzing big dynamic data. Inform Technol. 2016;58(4):195–205.
[58] Zaharia M, Das T, Li H, et al. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing (HotCloud); Boston, MA. 2012. p. 10–10.
[59] Vavilapalli VK, Murthy AC, Douglas C, et al. Apache Hadoop YARN: yet another resource negotiator. Proceedings of the ACM 4th Annual Symposium on Cloud Computing (SOCC '13); New York, NY, No. 5. 2013.
[60] Carbone P, Ewenz S, Haridiy S, et al. Apache flink: stream and batch processing in a single engine. Bull IEEE Comput Soc Tech Committee Data Eng. 2015;38(4):28–38.
[61] Tzoumas K, Metzger R, Ewen S. High-throughput, low-latency, and exactly-once stream processing with Apache Flink; 2015. Artisans [Internet]. Available from: https://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink/
[62] Chandy M, Lamport L. Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst. 1985;3(1):63–75.
[63] Fu TZ, Ding J, Ma RT, et al. DRS: dynamic resource scheduling for real-time analytics over fast streams. Proceedings of the IEEE 35th International Conference on Distributed Computing Systems; Columbus, OH. 2015. p. 411–420.
[64] Rychly M, Skoda P, Smrz R. Scheduling decisions in stream processing on heterogeneous clusters. Proceedings of the Eighth International Conference on Complex, Intelligent and Software Intensive Systems (CISIS '14); Washington, DC. 2014. p. 614–619.
[65] Zaharia M, Das T , Li H , et al. Discretized streams: a fault-tolerant model for scalable stream processing. Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing; Boston, MA. 2012.
[66] Eskandari L, Huang Z, Eyers D. P-Scheduler: adaptive hierarchical scheduling in apache storm. Proceedings of the Australasian Computer Science Week Multiconference, no. 26; Canberra, Australia. 2016.
[67] The Apache Software Foundation. Apache Mesos [Internet]. 2018. Available from: http://mesos.apache.org/
[68] Assuncao MD, Veith AS, Buyya R. Distributed data stream processing and edge computing: a survey on resource elasticity and future directions. 2017; arXiv:1709.01363v2.
[69] Carbone P, Ewen S, Fora G, et al. State management in Apache Flink: consistent stateful distributed stream processing. Proceedings of the VLDB Endowment, Vol. 10(12); Munich, Germany. 2017. p. 1718–1729.
[70] Venkataraman S, Panda A, Ousterhout K, et al. Drizzle: fast and adaptable stream processing at scale. Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17); Shanghai, China. 2017. p. 374–389.
[71] Chen F, Gu L, Jin H, et al. Lever: towards low-latency batched stream processing by pre-scheduling. Proceedings of the ACM Symposium on Cloud Computing (SoCC '17); Santa Clara, CA. 2017.
[72] Das T, Zhong Y, Stoica I, et al. Adaptive stream processing using dynamic batch sizing. Proceedings of the ACM Symposium on Cloud Computing (SOCC '14); Seattle, WA. 2014. p. 1–13.
[73] Liao X, Gao Z, Ji W, et al. An enforcement of real time scheduling in spark streaming. Proceedings of the Sixth International Green and Sustainable Computing Conference (IGSC); Las Vegas, NV. 2015. p. 1–6.
[74] Meng-meng C, Chuang Z, Zhao L, et al. A task scheduling approach for real-time stream processing. Proceedings of the IEEE International Conference on Big Data and Cloud Computing (BdCloud); Wuhan, China. 2014. p. 160–167.

[75] The Apache Software Foundation. S4-distributed stream computing platform [Internet]. 2013. Available from: http://incubator.apache.org/s4/
[76] Xu J, Chen Z, Tang J, et al. T-Storm: traffic-aware online scheduling in storm. Proceedings of the IEEE 34th International Conference on Distributed Computing Systems (ICDCS '14); Madrid, Spain. 2014. p. 535–544.
[77] Liu Y, Wang K, Yu Y, et al. A dynamic assignment scheduling algorithm for big data stream processing in mobile Internet services. Pers Ubiquitous Comput. 2016;20(3):373–383.
[78] Chen Q, Zhang D, Guo M, et al. SAMR: a self-adaptive MapReduce scheduling algorithm in heterogeneous environment. Proceedings of the 10th IEEE International Conference on Computer and Information Technology; Bradford, UK. 2016. p. 2736–2743.
[79] Dawei S, Hongbin Y, Shang G, et al. Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams. J Supercomput. 2017;74(2):615–636.
[80] Cardellini V, Grassi V, Presti F, et al. Distributed QoS-aware scheduling in storm. Poster session presented at: 9th ACM International Conference on Distributed Event-Based Systems (DEBS '15); Oslo, Norway. 2015..
[81] Pietzuch P, Ledlie J, Shneidman J, et al. Network-aware operator placement for stream-processing systems. Proceedings of the 22nd International Conference on Data Engineering (ICDE'06); Atlanta, GA. 2006. p. 49–49.
[82] Dabek F, Cox R, Kaashoek F, et al. Vivaldi: a decentralized network coordinate system. Proceedings of ACM SIGCOMM '04, Vol. 34(4); Taormina, Sicily. 2004. p. 15–26.
[83] Kephart JO, Chess DM. The vision of autonomic computing. Computer. 2003;36(1):41–50.
[84] Safaei A. Real-time processing of streaming big data. Real-Time Syst. 2017;53(1):1–44.
[85] Mohammadi S. Continuous query response time improvement based on system conditions and stream features [MSc thesis]. Iran: University of Science and Technology; 2010.
[86] Hoffmann M, McSherry F, Lattuada A. Latency-conscious data flow reconfiguration. Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR'18), No. 4; New York, NY. 2018.
[87] Del Monte B. Efficient migration of very large distributed state for scalable stream processing. Proceedings of the VLDB 2017 PhD Workshop; Munich, Germany. 2017.