Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions

XUNYUN LIU, Artificial Intelligence Research Center, National Innovation Institute of Defense Technology, China
RAJKUMAR BUYYA, The University of Melbourne, Australia

Stream processing is an emerging paradigm to handle data streams upon arrival, powering latency-critical application such as fraud detection, algorithmic trading, and health surveillance. Though there are a variety of Distributed Stream Processing Systems (DSPSs) that facilitate the development of streaming applications, resource management and task scheduling is not automatically handled by the DSPS middleware and requires a laborious process to tune toward specific deployment targets. As the advent of cloud computing has supported renting resources on-demand, it is of great interest to review the research progress of hosting streaming systems in clouds under certain Service Level Agreements (SLA) and cost constraints. In this article, we introduce the hierarchical structure of streaming systems, define the scope of the resource management problem, and present a comprehensive taxonomy in this context covering critical research topics such as resource provisioning, operator parallelisation, and task scheduling. The literature is then reviewed following the taxonomy structure, facilitating a deeper understanding of the research landscape through classification and comparison of existing works. Finally, we discuss the open issues and future research directions toward realising an automatic, SLA-aware resource management framework.

CCS Concepts: • **Software and its engineering** → *Cloud computing*; • **Networks** → Cloud computing; • **Computer systems organization** → Cloud computing;

Additional Key Words and Phrases: Resource management, stream processing, distributed stream processing systems, task scheduling

ACM Reference format:
Xunyun Liu and Rajkumar Buyya. 2020. Resource Management and Scheduling in Distributed Stream Processing Systems: A Taxonomy, Review, and Future Directions. *ACM Comput. Surv.* 53, 3, Article 50 (April 2020), 41 pages.
https://doi.org/10.1145/3355399

1 INTRODUCTION

With the popularisation of the Internet of Things (IoT), the number of intelligent devices used for monitoring, managing, and servicing has rapidly increased. These interconnected data sources

Authors' addresses: X. Liu, Artificial Intelligence Research Center, National Innovation Institute of Defense Technology, Beijing, 100071, China; email: liuxunyun@nudt.edu.cn; R. Buyya, The University of Melbourne, The Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, Parkville, VIC, 3010, Australia; email: rbuyya@unimelb.edu.au.

0360-0300/2020/04-ART50 $15.00
https://doi.org/10.1145/3355399

generate fresh data continuously, forming a large number, or a massive flow, of data streams that will eventually overwhelm the traditional data management systems. Meanwhile, the ever-growing data generation has been accompanied by the escalating demands for low-latency data processing. Time-critical applications such as fraud detection [103], algorithmic trading [1], and health surveillance [157] are gaining increasing popularity, all of which rely heavily on the low-latency guarantee to deliver meaningful results. The desire of fast data analysis gives birth to the emergence of stream processing, a new in-memory processing paradigm that allows for the collection, analysis, and visualisation of streaming data with only seconds or milliseconds latencies.

Unlike the traditional store-first, process-later batch paradigm, stream processing continuously consumes incoming data to provide immediate insights before the value quickly diminishes with time. The incoming data are handled upon arrival, with the results being incrementally updated while the data flow through the system. Presented with only limited resources to handle continuous inputs, stream processing has no random access to the whole stream. Instead, it installs processing logic over time- or buffer-based windows, conducting lightweight and independent computations over recently arriving data. In this way, the strict latency requirement can be met by proper workload balancing and processing parallelisation on a host of distributed resources.

Building a distributed streaming application from scratch is a tedious job and error-prone—developers have to write code for collecting input data, wiring processing logic, and reporting the value of insights with low latency. This is further exaggerated with the burdens of dynamic scaling and failure handling, which are common requirements for distributed computation. Over the recent years, various Distributed Stream Processing Systems (DSPSs) have been proposed to facilitate the development of streaming applications. From a structural perspective, a DSPS works as the middleware of a distributed system, offering unified stream management, imperative application programming interfaces (APIs), and a set of streaming primitives to simplify the application implementation. The state-of-the-art DSPSs, such as Apache Storm [150] and Apache Flink [16], further provide transparent fault-tolerance, horizontal scalability, and state management for the upper layer applications, while abstracting away the complexity of coordinating distributed resources. A typical streaming system is thus a three-tier structure comprising user-applications, DSPS, and the underlying infrastructure.

Though the adoption of DSPSs makes it easier to develop streaming applications, it remains a challenging and labour-intensive task to deploy a streaming system in a distributed environment satisfying certain Quality of Service (QoS) requirements with minimal resource cost. In this article, the context of deployment problem is mainly derived from the application provider's perspective, which we have broken down into three major research topics: (1) resource provisioning—determining the composition of the processing infrastructure, (2) operator parallelisation—configuring the degree of parallelism for streaming logic, and (3) task scheduling—deciding the placement of streaming tasks on distributed resources. The subtle interplay between these aspects plays a vital role for the deployed system to meet its functional and non-functional design requirements. There are other topics such as operator migration, state management, and operator graph optimisation that are commonly discussed within the streaming community. But they are excluded from our survey, as we assume that the state-of-the-art DSPS has provided the built-in mechanisms for the required functionalities, hence the deployment process does not involve user intervention in these aspects for resource management and task scheduling purposes.

Cloud computing has offered a scalable and elastic resource pool to enable a new level of freedom in system deployment. Its customers can unilaterally provision computing capabilities as needed through an automatic-measured, subscription-oriented model, where the monetary cost is billed on a pay-as-you-go basis. The advent of cloud computing also makes it harder to manage resources for streaming systems due to a combination of influencing factors, such as the sensitive

application requirements, dynamic workload characteristics, various cloud resource types, and diverse pricing models. Improper resource management and task scheduling directly affect the system performance on clouds. For example, over-provisioning and under-provisioning of resources lead to extra operational cost and Service Level Agreement (SLA) breaches, respectively. Acquiring resources from a suboptimal location introduces additional communication latency and network traffic, and inappropriate parallelisation of operators results in either overload streaming tasks or excessive overhead of context switching. Last but not least, misplacing streaming tasks to the underlying infrastructure leads to inefficient stream routing and resource contention that impair the system stability.

There have been quite a few surveys and taxonomies being conducted in the realm of distributed stream processing systems. Some of them have reviewed the whole stream processing landscape. Cugola et al. [31] wrote a seminal survey on information flow processing that aims to merge the results produced by the data stream processing model and the complex event processing model. Compared to our work, their survey stands at a higher viewpoint covering data and processing models, the language used to express the processing logic, and the runtime system architecture. Kamburugamuve et al. [76] conducted a survey on distributed stream processing systems with a focus on fault tolerance and comparison among different DSPS implementations. Hirzel et al.'s work [3] presents a catalog of optimisations to improve the performance of stream processing systems, but only part of the optimisation techniques, such as task scheduling and load balancing, are relevant to the deployment process for not changing the application graph and altering the processing semantics. Dayarathna et al. [34] investigated on system architecture characteristics of various event processing platforms, summarising the advancements made on open research topics such as event ordering, system scalability, event processing languages, and the use of heterogeneous devices. Their survey is on general data stream processing covering both systems and use cases, while our review has a narrower focus on deploying stream processing systems on cloud with SLA-awareness and cost-efficiency. Recently, Röger et al. [128] developed a classification of existing methods for both parallelisation and elasticity in stream processing systems. Though they have carefully examined the literature on parallelisation and elasticity in various aspects such as system type, programming model, and memory architecture, there is only a brief discussion on resource provisioning and task scheduling as part of the related work. In contrast, our review takes these two topics as the first-class citizens of resource management and provides a comprehensive overview and categorization of existing work with a taxonomy.

Some other surveys are conducted in the resource management and scheduling contexts, but each of them has a more specific focus in this area without holistically covering the deployment problem. Lakshmanan et al. [89] reviewed the various algorithms for task placement in data stream management systems, identifying a set of core placement design characteristics such as software implementation, algorithm structure, and considered metrics to help designers judge which placement strategy is best suited to a specific problem. This work was done a decade ago so there is a great need to timely review the research progress in the field of task scheduling. Zhao et al. [170] surveyed various types of stream processing systems and discussed the default methods for resource management in different DSPSs. Dias de Assunção et al. [38] surveyed the state-of-the-art stream processing engines with a focus on the enabling mechanisms for resource elasticity. Hummer et al. [67] also provided an overview of stream processing and explained the key concepts pertaining to runtime adaptivity and cloud-based elasticity, but SLA-aware resource management is not included in their survey. There are also some surveys that have discussed the patterns and infrastructure to run stream processing systems elastically [54, 55, 61, 123, 133], but they emphasise more on the resource provisioning problem and lack sufficient discussion on operator parallelisation and task scheduling.

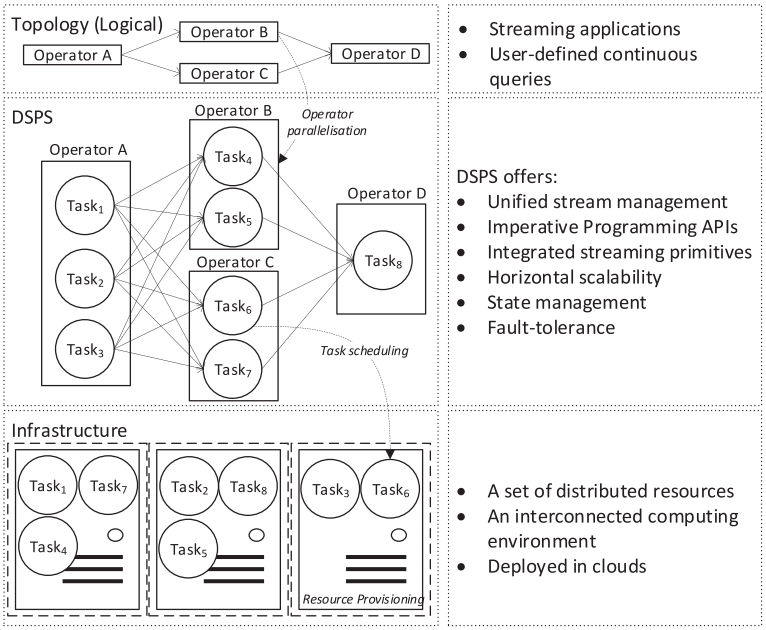

Fig. 1. The sketch of the hierarchical structure of a streaming system.

As the research in this area advances, there is a long overdue effort to define the scope of the resource management and scheduling problem in the stream processing context, then comprehensively analyse the recent progress and identify the main challenges to achieve better SLA-awareness and cost-efficiency. In this article, we aim to bridge this gap by **proposing** a taxonomy of resource management and scheduling techniques, surveying existing work with regard to the taxonomy architecture, and discussing the open issues and challenges worth pursuing in the future.

The rest of the article is organised as follows: We first introduce the hierarchical structure of a distributed streaming system as background, using it to organise the research topics that are involved in the deployment process and covered in this review. Then we present a taxonomy of resource management and scheduling to classify the key properties of existing work. In light of the taxonomy, the surveyed works are mapped into different categories for better comparison of their strengths and weaknesses. A thorough analysis of existing work also sheds light on the promising future directions toward an SLA-aware, cost-efficient, and self-adaptive resource management and scheduling framework, which we discuss before closing this article.

2 BACKGROUND

The resource management and task scheduling problem is part of the deployment process to ensure that the pre-defined service level agreements are met and that the resource cost is minimised. This section introduces the motivation and challenges of the targeted problem, and Appendix A.1 outlines how SLA-awareness can be achieved through a self-managing and self-adaptive resource management process.

To better understand the problem scope, Figure 1 presents the hierarchical structure of an example streaming system. Sitting on the topmost level is the abstraction of the streaming logic, which in this case consists of four operators standing on incoming data streams. These inter-connected operators constitute a Directed Acyclic Graph (DAG) called *topology*, representing a streaming

application that produces incremental results on-the-fly unless being explicitly terminated. Each operator encapsulates certain streaming logic such as data filtering, stream aggregation, or function evaluation, while the edges denote the data paths between operators as well as the sequence of operations conducted on the data streams. In most cases, the DAG of operators has been properly defined upon the completion of application development. Once entering the deployment phase, one has to decide where and how these streaming logic are executed in a live distributed environment to cater for the continuous and possibly fluctuating workload with SLA-awareness.

A Distributed Stream Processing System (DSPS) is positioned in the middle of the system structure, executing the DAG of operators in a data-parallel and pipelined manner to provide high-throughput, low-latency stream processing capabilities. Specifically, DSPSs expose a set of imperative programming APIs and streaming primitives to developers, encapsulating low-level implementation details such as stream routing, data serialisation, and buffer management in a unified streaming model. Developers can thus focus on the implementation of streaming logic without having to reinvent the wheels for routine data management. DSPSs also provide abstractions for parallel and distributed computing, allowing applications to exploit horizontal/vertical scalability and fault-tolerance without code changes. During the deployment phase, the parallel operators in the topology can scale with a given parallelism degree, generating multiple replicas, known as *tasks*, to execute simultaneously on top of distributed resources. As illustrated in Figure 1, *Operator B* is parallelised into $Task_4$ and $Task_5$ due to *operator parallelisation*. Afterward, a process called *task scheduling* dynamically assigns the streaming tasks to distributed resources, e.g., $Task_6$ of *Operator C* is mapped to the computing node at the right end of Figure 1 for execution. Conveniently, the DSPS guarantees the semantic correctness of parallelisation with built-in mechanisms like automatic stream splitting and tuple tracking. Heinze et al. [55] classified the existing DSPSs into three generations, among which we mainly focus on the third generation, which is highly distributed and even applicable to heterogeneous environments such as edge and fog clouds. Notable implementations falling into this generation include S4 (Simple Scalable Streaming System) [2], Apache Storm, Twitter Heron [87], Apache Flink, Samza [117], Spark Streaming [167], and so on.

The underlying infrastructure level represents the physical view of a stream processing system, which is an interconnected computing environment created by a process called *resource provisioning*. In this article, we only consider the Infrastructure-as-a-Service (IaaS) model for provisioning resources in clouds. This model visualises the physical infrastructure as separate service components such as computing, storage, and network, where users can deploy their applications with the finest control over the entire software stack, including operating systems, middleware, and applications. There are also streaming services available in the form of the Platform as a Service model and the Software as a Service model. Notable examples include Silicus,[1] Google Dataflow,[2] and Microsoft Azure Stream Analytics.[3] However, the deployment of streaming applications on these services is usually managed by the service owner rather than the application provider, making it harder for the stakeholders to directly manage resources for boosting performance and cost-efficiency.

As shown in Figure 1, deploying a streaming system can be regarded as a decision and configuration problem that constructs the hierarchical system structure in a distributed environment, where the higher tier is mapped to and hosted on the lower tier to be concrete and runnable. The primary motivation of having a resource management and scheduling framework is to free the application providers from the burden of manual tuning. By applying a collection of profiling,

[1] https://www.silicus.com/iot/services/stream-processing-and-analytics.html.
[2] https://cloud.google.com/dataflow/.
[3] https://azure.microsoft.com/en-gb/services/stream-analytics/.

modelling, and decisioning techniques, the framework, which is discussed in Appendix A.1, can be trusted to ensure the deployed system meet its SLA requirements with minimal resource consumption.

The scope of resource management and scheduling is broken down into main three sub-research topics. We explain each topic by highlighting its peculiar problem domain, as well as discussing the issues and challenges faced by developers to achieve SLA-awareness and cost-efficiency.

Resource Provisioning. Resource provisioning describes the activities to estimate, select, and allocate appropriate resources from the service provider to constitute the interconnected stream processing environment.

- Resource estimation: estimating the type and amount of resources needed by the system to meet its performance and cost targets articulated in the SLA. Such estimation can be derived from the analysis of historical data as well as the prediction of future workload, but its accuracy is often affected by the instantaneous, unexpected fluctuation of inputs and system performance variations due to the dynamic nature of data streams.
- Resource adaptation: the real resource demands can fluctuate along with the varying workload, or remain vague and unclear even after the system is brought online. Finding the right point in time to scale in/out and choosing the right adaptation scheme remains a huge challenge. In addition, the profitability of adaptation is affected by a number of factors such as the selected billing model and the geographical distribution of resource pools. Take the latter case as an example, the non-negligible network latency must be taken into consideration when performing system adaptation in a distributed manner [17, 20, 121].

Operator Parallelisation. Operator parallelisation divides a parallel operator into several functionally equivalent replicas, each handling a subset of the whole operator inputs to accelerate data processing.

- Parallelism calculation: This would require accurate profiling of stream workload and probing the processing capability of each task. The details of the infrastructure also matter—the number of cores/threads in a CPU confines the maximum degree of runtime parallelism and the hardware implementation determines the cost of thread scheduling and context switching.
- Parallelism adjustment: Over-parallelisation and under-parallelisation can occur at runtime as a result of workload change or resource adaptation. It remains a major challenge to monitor and profile streaming tasks at a fine-grained level to reveal the true performance bottleneck of the application. Another challenge is transparent state management during adjustment—stateful operators need to repartition and migrate their states properly among the constituent tasks to make parallelism adjustment transparent to developers.
- Balancing data source[4]/sinks[5]: The parallelism degree of an operator reflects the adequacy of access to the distributed resources. While making parallelisation decisions, the balance between data sources and data sinks needs to be fine-tuned as their performances are correlated due to the producer and consumer communication model in the streaming system. An overly powerful data source may cause severe backlogs in data sinks, whereas an inefficient data source would starve the subsequent operators and encumber the overall throughput [100].

[4]A data source is an operator responsible for injecting data into the stream processing graph.
[5]A data sink is a peripheral operator that does not have any outgoing edges and only consume data in the stream processing graph.

Task Scheduling. Task scheduling dynamically maps streaming tasks to horizontally/vertically scaled resources, such that data streams are partitioned and processed at different locations simultaneously and independently. In this review, we assume that the load balancing of stream routing relies on the DSPS to properly partition data streams among the streaming tasks belonging to the same operator. In addition, we assume that there is a state migration mechanism, like the one in Reference [22], to handle the migration of internal task state to the node where the stateful task is being rescheduled.

- Minimising inter-node communication: inter-node communication is much costlier than intra-node communication as the former involves time-consuming operations such as message serialisation and network transfer. It is therefore preferable to place communicating tasks on the same node as long as it does not cause resource contention. If the infrastructure consists of geographically distributed resources, then it becomes even more prominent to reduce large data transmissions on remote and error-prone data links with limited bandwidth.
- Mitigating resource contention: one of the leading causes of performance deterioration is the competition for computational and network resources among collocated tasks. There is a great interest in designing a resource-aware scheduler that makes sure the accrued resource demands of collocated tasks do not exceed the node's capacity.
- Performance-oriented scheduling: the scheduling of tasks should be optimised toward the specific application performance targets defined in the SLA, regardless of the interference brought by workload fluctuations, virtual machine (VM) performance variations, and the multi-tenancy mechanism at the infrastructure and the DSPS tier.

Task scheduling for stream processing systems is similar to workflow scheduling for batch processing systems—both of them are concerned with the assignment of tasks to the previously provisioned resources. However, they also differ from each other as the objects being scheduled exhibit distinct properties. Streaming tasks possess indefinite lifespan and share a great deal of intercommunication due to the producer and consumer model inherent in stream processing, while batch tasks only exist for a limited time with the dependency of execution hinging on the sequence of completion rather than the continuous and intermediate streams. These differences are naturally reflected in the scheduling process. The former involves real-time decision making considering the dynamic nature of input data, such as the varying intensity and complexity of the data flow, whereas the latter can be done prior to the processing of batch jobs based on the *priori* knowledge of data, tasks, and the execution environment.

It is also common that the data streams being transmitted in DSPSs are handled in batches for performance reasons. Due to the strict latency constraint, batches in DSPSs are rather small and processed at small intervals, which gives birth to a new concept called micro-batch. Comparing to the canonical streaming model that handles each new piece of data when it arrives, the micro-batch model divides the stream into small batches of a fixed duration and processes them at individual batch windows, gaining better reliability and exactly once semantics at the cost of higher latencies. The micro-batch model also plays a vital role in striking a balance between throughput and latency—the two most significant performance indicators in stream processing. For instance, enlarging the batch size in micro-batching may deteriorate end-to-end latencies due to the increased buffer filling time, but it helps improve batch throughput by reducing the frequency of small communication calls. From the scheduling perspective, the micro-batch model introduces a new level of scheduling called job scheduling—with each batch considered as a job, the job scheduler receives the periodically generated jobs and decides when and how to schedule them in DSPS for execution. There is also a number of scheduling parameters such as batch size, job parallelism and resource shares among jobs to be tuned to cope with the variations in the workload and system

conditions. However, due to the page limit, the scope of our review is confined to task scheduling alone, which applies to both of the canonical streaming model and the micro-batch model.

Another topic on-trend in big data processing is the Lambda architecture designed to incorporate both batch and stream processing methods in a uniform manner [16, 117]. It describes systems that consist of three layers: a batch layer for processing all available data at rest (typically on persistent storage), a speed layer for processing the most recent data in motion, and a serving layer for responding to user queries in low-latency on an ad hoc basis. This sort of architecture aims to balance latency, throughput, and fault-tolerance by providing comprehensive and accurate views of batch data, as well as rough results and quick insights of online data through continuous stream processing. Essentially, the principle of lambda architecture implies a loosely coupled design—each of the batch and streaming sides maintains a different code base and processes data independently from different paths. Only at the query time are the results from both systems stitched together to produce a complete answer. Our review applies specifically to the speed layer of the lambda architecture, in which a stream processing system is employed to provide quick analysis of big data and can benefit from the various resource management techniques surveyed in this article.

3 TAXONOMY

Figure 2 presents a taxonomy of resource management and scheduling in distributed stream processing systems. In Section 2, we have broadly broken down the topic of interest into main three sub-research topics—resource provisioning, operator parallelisation, and task scheduling. This taxonomy explores the sub-research topics further, covering seven specific aspects to allow better comparison of existing work with similar research targets and method properties. Structuring the taxonomy this way echoes the challenges and issues identified in Section 2, clearly defines the scope of our survey, and demonstrates the necessity of using a combination of resource management and task scheduling techniques to achieve SLA-awareness and cost-efficiency. The stream processing community can benefit from the classification of current practice, and the developers can make use of the identified common patterns to build a system respecting given target SLAs.

- Resource Type: the various resource types involved in the resource management process to compose the stream processing infrastructure.
- Resource Estimation: the estimation and modelling of resource costs for a streaming system to satisfy its SLA requirements.
- Resource Adaptation: the adaptation of resource allocation to the changes of workload volume and application performance.
- Parallelism Calculation: the profiling and calculation of parallelism degree for the parallel operators in the application topology.
- Parallelism Adjustment: the adaptation of operator parallelism in response to workload variations and internal system changes.
- Scheduling Objective: the various objectives of task scheduling and the rationale behind these objectives to achieve the overall deployment target.
- Scheduling Methods: the various methods used for task scheduling.

Though our taxonomy has broken down the topic of interest into seven different aspects, this only serves as a roadmap to review the literature of resource management and task scheduling from different perspectives rather than specifying the boundaries of research. As discussed in Appendix A.1, the activities of resource management and scheduling are tightly correlated—they are often conducted in a bundle to fulfill a holistic deployment target. For example, a complete resource provisioning cycle consists of three steps—selecting particular resource types (Resource Type), estimating requirement of resource allocation (Resource Estimation), and adapting resource

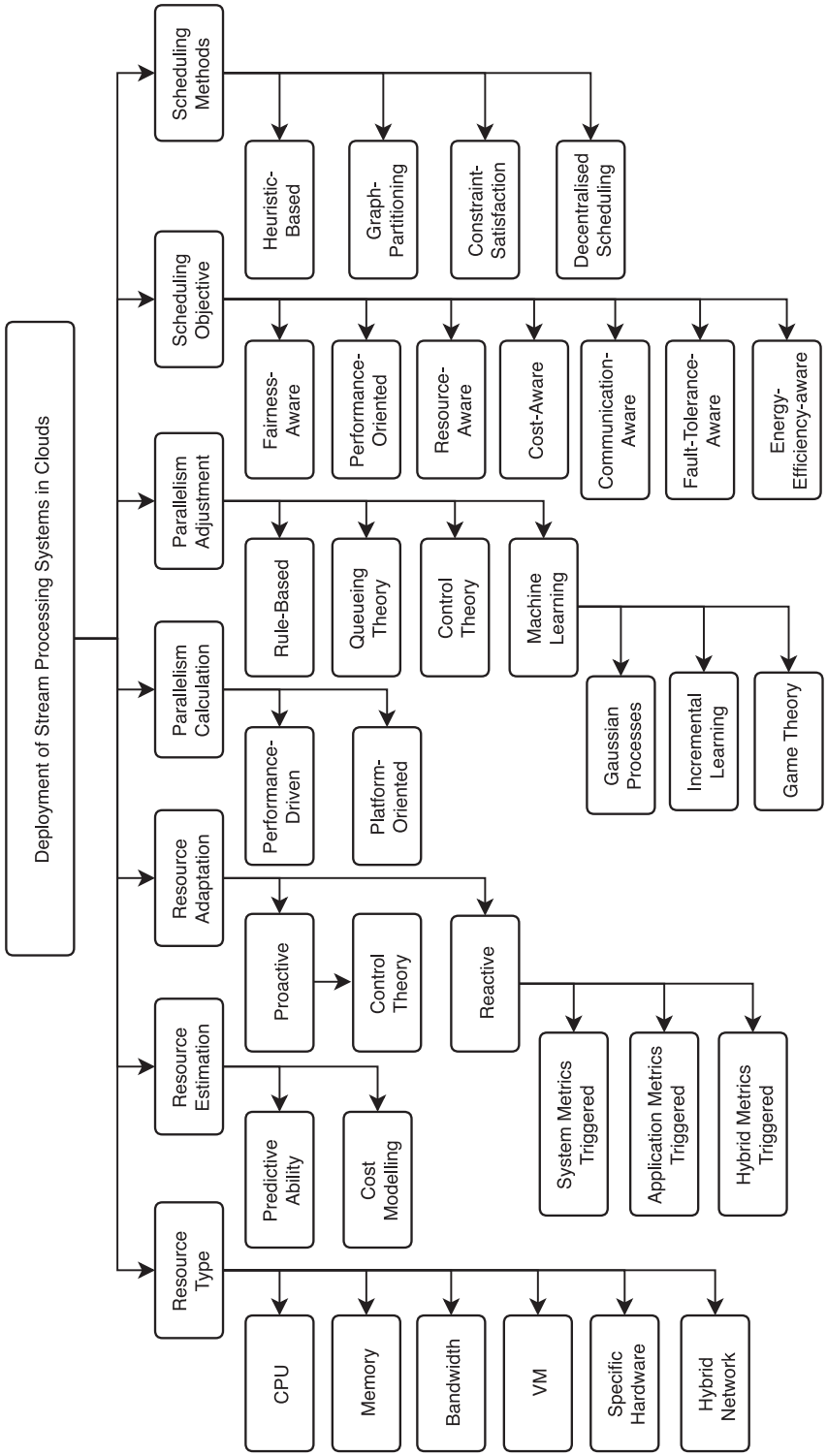

Fig. 2. The taxonomy of resource management and scheduling in distributed stream processing systems.

allocation in response to runtime changes (Resource Adaptation), where the former step is regarded as a prerequisite for the latter. Since a relevant research may stretch across multiple aspects for completeness, it can be covered multiple times in the following sections (Section 4 ∽ Section 10) with a different focus of the taxonomy.

4 RESOURCE TYPE

Resource describes any physical or virtual component of limited availability within a computer system. However, depending on the actual context, the same term could contain diverse meanings and refer to various resource types at different levels of abstraction. For deployment in IaaS clouds, resource generally refers to the computing and network facilities that are available to rent through usage-based billing, such as VMs, IP addresses, and Virtual Local Area Networks. However, when a streaming system is to be deployed in a more hybrid and geographically distributed environment, the concerned resource types also include other infrastructural components such as specific hardware and hybrid networks.

In this section, we identify the various resource types involved in the deployment and resource management process. It is worth noting that the storage resources, such as block, file, or object storage, are omitted in our classification due to the rare discussion in the literature. This is credited to the fact that saving stream data to an off-site storage system is often prohibitive, which would block the dynamic data flow and cause unsustainable processing latency.

4.1 Resource Abstractions

Resource abstractions, such as CPU, memory, and network bandwidth, quantify the resource requirements of a streaming system, regardless of the hardware differences at the infrastructure layer. In addition, network latency is a major constraining factor that determines if the streaming application can fulfill its low-latency SLA. From the end-users' perspective, the measurement of resource abstractions is intuitive and straightforward. CPU resources can be counted by the number of used CPU cores, with loads measured by Million Instructions Per Second (MIPS) or percentage utilisations. Memory usage refers to the amount of memory currently in use and is quantified by Megabytes (MB). Network bandwidth consumption corresponds to the average rate of successful data transfer through a communication path, which is gauged by the unit of Megabytes per second (MB/s) or Kilobytes per second (KB/s). Network latency indicates any kind of delay that happens in data communication over a network and is often measured by milliseconds.

However, ignoring the particularity of the underlying infrastructure also means that the measurement of resource abstractions reflects the general system state rather than yielding actual resource provisioning plans. The results would be susceptible to countless modelling nuances and hardware discrepancies. Instead of being used to directly construct the infrastructure, resource abstractions are more commonly seen in rule-based approaches (Section 6.2) to approximate the resource cost and shed light on the direction of adjustments.

4.2 Virtual Machines

Virtual machine (VM) is an emulation of a computer system customisable to meet the specific user needs. In a cloud environment, virtual machine is the most common resource type that encapsulates the computing power and serves as the host of streaming tasks in a distributed environment.

Provisioning VMs from a particular cloud platform is a mixed problem of considering the VM price model, the location of data centres, and the network capacity of inter-connections. The actual VM configurations and placement are determined by the specific computation and communication needs of the streaming system to meet its performance and cost SLA. For the generality of discussion, this survey also includes resource management techniques that originally apply to

the on-premise cluster environment, as the proposed resource estimation and adaptation methods would also benefit the VM management in clouds to prevent resource leaks and contentions.

Containers are similar to VMs in terms of encapsulating virtualised resources to host the streaming tasks, but they are more lightweight as the isolation properties are relaxed to share the operating system among the applications. It is increasingly common to deploy streaming systems over containers rather than VMs to exploit the benefits of portability and efficiency. In these cases, the underlying containers can be considered as a special type of VMs from the resource management and task scheduling perspective. Many of the existing resource management techniques still apply to the container cloud with little modifications required.

4.3 Specific Hardware

The infrastructure of streaming systems may require specific hardware to boost performance, improve manageability, and deal with particular streaming scenarios. Due to the scarcity of supply and the indispensability of functionality, provisioning of these critical resources is often prioritised over other common computing and network resources in clouds.

Chen et al. [29] proposed a GPU-enabled extension on Apache Storm, exploiting the massively parallel computing power of the Single Instruction Multiple Data (SIMD) architecture to accelerate the processing of stream data. Similarly, Espeland et al. [42] processed distributed real-time multimedia data on GPUs with support for transparent scaling and massive data- and task-parallelism.

FPGA is reconfigurable hardware designed to enable hardware-accelerated computations. The use of FPGA as central data processing elements allows exploiting low-level data and functional parallelism in streaming applications. To facilitate the application of FPGA for stream processing, Auerbach et al. [5] presented a Java-compatible language as well as the associated compiler and run-time system to integrate the streaming paradigm into a mainstream programming environment. Neuendorffer et al. [116] from Xilinx discussed the design tools required for the fast implementation of streaming systems on FPGAs, and Sadoghi et al. [131] investigated how to map multiple streaming applications to FPGA hardware using Hardware Description Language (HDL) code.

Specific hardware can not only be found at the centralized cloud but also at different locations of the network to facilitate the timely processing of stream data. In some use cases, deploying the streaming system requires specific sensors to collect input data or monitor the current processing state such as network transmission and power consumption. For instance, data collection sensors are employed by Zhu and Vijayakumar [153, 173] to aggregate stream data from the satellites and environmental monitoring facilities in real time. Kamburugamuve et al. [75] proposed a hybrid platform to connect smart sensors and cloud services, with the data processing logic deployed in the centralised cloud servers to enable new real-time robotics applications such as autonomous robot navigation. Traub et al. [151] optimised communication costs on sensor networks by sharing sensor reads among streaming applications, so that the amount of data transfer is reduced by a combination of data stream sampling and tailoring techniques. Also, power meters such as Watts Up are employed by Shen et al. [139] and Mashayekhy et al. [110] in their streaming systems to get live power readings from the host machines.

4.4 Hybrid Network

Traditionally, streaming systems are deployed in a single cluster or cloud environment as most of the data streams to be processed are collected from web analytic applications. However, there is an ongoing trend that the deployment migrates to a more heterogeneous and geographically distributed setting to process the huge data streams generated by the IoT applications. In this process, novel network elements and hybrid network structures have been employed to enhance the infrastructure connectivity and create new application paradigms.

The emergence of programmable networking hardware and the expressive data plane programming languages motivate the idea of in-network computation [8, 9, 96, 132]. In-network stream processing delegates part of the computing operations to the network devices such as switches and smart Network Interface Cards (NIC), to reduce the increasing data traffic to data centres. However, this brings new challenges to the design of the scheduler to decide what type of computation can be done in-network. The streaming applications are sensitive to the varying network performance yet there are strict end-to-end latency constraints to respect, so the scheduler must make a careful selection of streaming tasks that can work with the limitations of the network architecture and the confined computing power of programmable devices. As well, the scheduler needs to consider maintaining scalability with the rapidly increasing communication cost as the stream data are routinely moved in many-to-many patterns.

Collaborative Fog, Edge, and IoT networks are also gaining popularity in stream processing for the ability to offload a substantial amount of control, computation and management workload to the network gateways close to data sources, thus reducing data transmission and bandwidth consumption. Papageorgiou et al. [119] identified that the low latency requirement is often challenged at the edge of the application topology due to the frequent communication with external IoT entities, so they built new decision modules to place selected tasks on edge devices at runtime using resource descriptors. Hochreiner et al. [62] discussed the distributed deployment of streaming applications over a hybrid cloud, with a threshold-based resource elasticity mechanism to deal with the variation of IoT streams. Cardellini et al. [20] also investigated distributed deployment of streaming systems over a geographically distributed Fog infrastructure, in which they focused on the design and implementation of a QoS-aware and decentralised scheduling policy. A distributed IoT network developed by Ralf et al. [122] tackles aggregation and processing of streaming data in smart city applications, which is capable of enriching input streams with semantic annotations and utilising stream reasoning techniques to allow real-time intelligence with event detection.

Mobile devices have also taken part in the network infrastructure of a streaming system to move computation closer to the data sources. To deploy stream processing application directly on smartphones, Wang et al. [158] proposed a new check-pointing method to mask the simultaneous failure of mobile devices and employed a segmented, UDP-based data transmission method to reduce the cellular network overhead. Similarly, Morales et al. [114] relied on mobile devices to pre-process data streams, and they also proposed a new check-pointing method that is both connectivity-aware and energy-aware. Yang et al. [165] discussed how to enable mobile devices to work in partnership with VMs provisioned in clouds, with a focus on the dynamic partitioning of data streams between mobile devices and data centres to achieve higher throughput and scalability.

However, High-performance Computing (HPC) network has also been employed in stream processing to enable advanced interconnectivity and better scalability than the conventional Ethernet connection. Recently, Kamburugamuve et al. [77] discussed the use of Infiniband and Intel Omni-Path to improve the performance of stream processing applications, where a new Storm extension is proposed, exploiting the native function of high-performance interconnects to achieve significantly lower latencies and improved throughputs.

5 RESOURCE ESTIMATION

Based on the information retrieved, recorded, or derived from the present and the past system states, resource estimation calculates the minimal amount of resources required by the streaming system to fulfill its SLA. The accuracy of resource estimation determines the cost-efficiency of resource provisioning, which plays a key role in a quick converge to optimal deployment and avoiding over- and under-resource utilisation.

Our taxonomy covers the following characteristics of a resource estimation method:

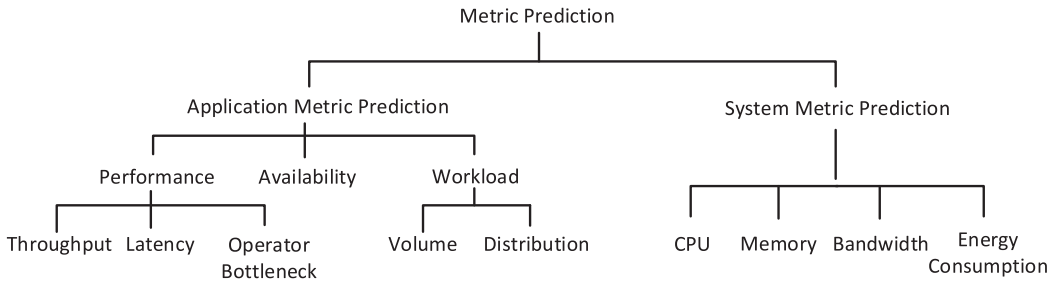

Fig. 3. The classification of predicted metrics used for resource estimation.

- Predictive Ability: whether the resource estimation method can predict future application and system metrics, such as workload size, resource utilisation, and application performance.
- Resource cost modelling: how it models the resource costs based on the predicted or collected metrics, and what criteria in SLA determine the minimal amount of resource requirements.

5.1 Predictive Ability

Prediction of future application and system metrics allows active speculation of future resource demands rather than assuming a constant resource consumption pattern. Figure 3 illustrates the classification of predicted metrics based on the level of which they are collected from the software stack. Metrics of different granularities contain different information and thus contributing to resource estimation in different ways. The prediction of system metrics normally leads to a direct estimation of future resource requirements, which is oblivious to the particularity of the hosted applications; whereas the prediction of application metrics leads to an indirect estimation of resource demands, which requires further resource cost modelling to suggest the minimal resource requirement without violating the SLA requirements.

From the methodology perspective, time series analysis and queueing theory are identified as the two prominent approaches for metric prediction.

5.1.1 Time Series Analysis. A time series is a sequence of data records collected at successive points in time, and time series analysis is an umbrella term that describes a variety of models and methods on time series to find repeating patterns in the historical data. In the context of stream processing, time series analysis can work on the past system resource usages and application metrics, leading to direct and indirect estimation of future resource requirements, respectively.

Direct Resource Estimation by Predicting System Metrics. CloudScale [139] is an elastic resource scaling system built on Xen hypervisor[6] that directly predicts the short-term resource demands based on the recent history of system metrics. They adopted a hybrid time series analysis approach combining both Fast Fourier Transform (FFT) and a discrete-time Markov chain to balance between high estimation accuracy and low overhead. The light-weight FFT is tried first for fast identification of repeating patterns in the previous time series. If not found, then the heavier Markov chain model performs multi-step analysis on the metric history to provide coarse-grained and long-term resource estimations.

The same approach is also seen in the group's previous work [171], with more details revealed on the prediction process. The application of FFT identifies the dominant frequencies of variation in

[6] https://www.xenproject.org/.

the observed resource-usage time series, followed by a discrete-time Markov chain model that unveils the deeper-hidden patterns through calculating the feature value distribution of the collected resource metrics. The combination of these two methods leads to a fast yet accurate estimation model, provided that there are patterns concealed in the resource usage history.

OrientStream [155] is a recent work on dynamic resource provisioning of stream processing systems. It features an online resource prediction module that employs an ensemble regression model on the past system metrics to suggest future resource usages. The prediction process is essentially a weighted vote of four independent regression models, reaping the benefit of reducing the overall Relative Absolute Error.

Dai et al. [33] presented VM provisioning as a multi-objective optimisation problem, which they solve with an auto-regressive model that learns and predicts the utilisation of each VM as well as the bandwidth consumption between routers. With further consideration on power management, Liu et al. [97] applied deep reinforcement learning over a linear combination of system metrics such as total power consumption, VM latency, and reliability metrics to synthetically predict future system states. Based on the forecast, a hierarchical resource provisioning model is proposed that saves energy consumption without significantly impacting application performance and availability.

Indirect Resource Estimation by Predicting Application Metrics. Time series analysis can also work on the historical application metrics to indirectly suggest the future resource demands with the help of resource cost modelling.

The first category of works predicts the future state of operators. Hidalgo et al. [60] applied a Markov chain model on the workload time series to predict whether an operator's future state would be overloaded, underloaded, or stable. Based on the state predictions, resource cost is modelled by checking the minimal amount of resources needed for the placement of tasks. Kombi et al. [86] adopted a similar method to forecast operator bottlenecks, which circumvents the rigorousness of queuing theory while still being able to estimate resource demands at the operator level. Ottenwälder et al. [118] achieved location-aware adaptation of operators with a query predictor. The proposed algorithm can reduce the latency during operator graph switches by configuring the graph deployments in advance, where the predicted operator states such as the possible location of the focal point and the future processing interests are utilised to guide the deployments.

The second category tries to predict future workload. Balkesen et al. [7] applied exponential smoothing on the periodic observations of the input stream rate to forecast the volume of future workloads. Their rate-forecasting heuristic solves the bin packing problem formulation, suggesting the future resource usages based on the stream distribution and the placement of operators. Analogously, Ishii et al. [70] employed Sequentially Discounting AutoRegression to predict future input rates. They formulated an optimisation problem on resource provisioning and solved it with linear programming to find the minimal resource requirement without violating the application latency SLA. Hoseiny Farahabady et al. [64] also predicted the changes in the input traffic with an Auto Regressive Integrated Moving Average model, which lays the foundation for a resource provisioning algorithm that causes less QoS detriments over all available servers.

Mayer et al. [112] predicted the workload distribution and its parameters with a hybrid approach of distribution moments and maximum likelihood method. The predicted workload distribution feeds into the calculation of operator parallelism and then sheds light on the resource cost by counting the number of processor cores required for task execution. Imai et al. [69] trained a linear regression model on the performance data collected in an experimental environment, to predict the maximum sustainable throughput of the streaming application running on a larger number of VMs. Therefore, the cost model is built by directly linking the desired application performance with the number of VMs provisioned in the infrastructure.

5.1.2 Queueing Theory. Queueing theory is a set of mathematical models studying the waiting lines and queues to describe or predict the waiting time and queue lengths. In stream processing, queueing theory is often applied to application performance metrics—especially operator latency—to shed light on the possible data flow bottlenecks.

By modelling the operator as a $G/G/1$ queueing system, De Matteis et al. [35] regarded each operator task as a single server queue where both inter-arrival times and service times have a general distribution. The Kingman's formula is then used to approximate the mean waiting time at the operator level, which sheds light on the runtime adaptation of the number of used cores as well as the CPU frequency. The same modelling and solving technique are also found in a variety of literature [25, 36, 37, 91, 104], which proves that the Kingman's formula is widely accepted for latency modelling because of its accuracy and generality applying to arbitrary distributions of the inter-arrival time and service time.

Differently, Hoseiny Farahabady et al. [64] modelled the operator as a $G/G/k$ queue (k is the number of processors for the target operator) and employed Allen-Cunneen approximation to give an upper-bound of the sojourn time experienced by each tuple. Since there is no exact formula known for the $G/G/k$-model, Allen-Cunneen approximation provides asymptotically exact results under heavy traffic and is particularly suitable for streaming applications with highly utilised operators. Fu et al. [46] formulated the operator as a $M/M/k$ system, where M indicates Poisson distribution for arrival and Exponential distribution for service time. Accordingly, Erlang formula is applied to estimate the expected value of the total tuple sojourn time in the application. This is different to the $G/G/1$ and $G/G/k$ modelling at the operator level as the whole application topology is modelled as a Jackson open queueing network, which increases the rigorousness of the queueing model but is capable of providing more accurate latency estimations for the whole application if the model assumptions are met.

5.2 Resource Cost Modelling

With the domain knowledge of stream processing, a resource cost model summarises the various metrics collected at the runtime stack, suggesting an overall estimation of resource requirements for the streaming system to satisfy its particular SLA requirements.

Modelling resource costs requires taking the application logic and the desired deployment target into consideration. Different operators may exhibit different resource usage patterns. For example, a filtering operator shows a pattern of resource consumption linear to its workload volume, while a window operator exhibits periodical resource requirement peaks as the window slides or executes. When the deployment of applications is tuned toward higher reliability and availability, extra resources are provisioned for fault-tolerance or serving as headroom to confine the utilisation rate of each node in certain bounds. When the deployment emphasises cost-efficiency and lower runtime overheads, the placement of tasks is consolidated to as fewer nodes as possible to reduce resource usages and inter-node communications. Proper resource cost modelling is the key to deal with these variations and suggest the overall resource requirements accordingly.

For the convenience of discussion, our taxonomy categorises different resource cost models based on the intended SLA optimisation, i.e., which SLA requirement is more critical to determine the system resource cost in general.

Minimal Cost Model. This model intends to achieve the targeted performance requirement with minimal resource cost and provisions no spare resources to improve reliability and availability. Bin packing is the most common strategy to model the minimal resource cost based on the compact task placement. Setty et al. [138] used bin-packing formulation to determine the minimal number of VMs needed by the placement of topic-subscriber pairs, with a greedy heuristic to optimise

cost while respecting the constraint that the application communication must not exceed the VM bandwidth capacity. Heinze et al. developed FUGU, an elastic data stream processing prototype, to evaluate different scaling policies [58] and optimise the scaling parameters [59]. In both works, the resource requirements are estimated using a bin-packing model solved by a First Fit and Best Fit heuristic. Balkesen et al. [7] employed bin packing to dynamically re-assign data streams to different nodes, making runtime adjustments to the previous round of stream assignment rather than optimising from scratch to balance between result optimality and the overhead of stream redirections. Shukla et al. [141] proposed a model-based approach to offer reliable estimates of the resource allocation required, which is essentially a two-dimensional bin-packing problem (CPU and memory) solved by best-fit packing. Bin packing is also employed by Liu et al. [98, 99, 101], Xu et al. [163], Nardelli et al. [115] and Ghaderi et al. [50] to suggest minimal resource cost under a certain SLA requirement.

Reliability-oriented Model. This model provisions additional resources for state management and failure recovery. Madsen et al. [108, 109] proposed a Storm extension that replicates the same operator state across different nodes, allowing faster state migration in failure recovery and the scaling of stateful operators. The resource cost is thus calculated by the needs of state management to maintain semantic correctness and fault-tolerance. Similarly, Castro Fernandez et al. [23] designed a set of state management primitives to expose internal operator states to the DSPS for transparent failure handling and scaling. This leads to a resource cost model built on state management with extra computation and communication overhead introduced by failure recovery and periodical state check-pointing. Koldehofe et al. [85] proposed a rollback-recovery scheme with low run-time overhead, where the state information is incrementally replicated at preceding operators to reduce the resource consumption required for providing reliability guarantees.

Contention-aware Model. Model of this type permits certain resource allowances to handle random workload bursts when needed. Hoseiny Farahabady et al. [64] proposed a resource cost model that tracks and confines the CPU utilisation level of each node within an accepted range, and a similar approach is also found in Thamsen et al.'s work [148]. To limit the memory usage and CPU consumption within a certain bound, Cammert et al. [15] proposed a cost model to estimate resource utilisation of continuous queries based on the stream characteristics such as the average inter-arrival time and the average validity of tuples. The proposed fine-grained cost model is customised to a variety of operator types and streaming logic, making it possible to even quantify the impact of (re)optimisations on query plans.

Load-balancing-oriented Model. This model focuses on the fair utilisation of available resources and is the opposite of the compact task placement, which is commonly seen in the minimal cost model. Fischer et al. [43], Eskandari et al. [41], and Jiang et al. [73] regard the operator placement as a graph partitioning problem, so that they explicitly spread the streaming tasks across all available resources at the infrastructure for better load balancing. This cost model is also employed by the round-robin scheduler that is used as default by a variety of DSPSs, which favours even load distribution over participating computing nodes.

Distribution-based Model. Also, depending on the nature of the cost model, the result of resource requirement may be a probability distribution rather than a definitive value. Khoshkbarforoushha et al. [82, 83] employed Mixture Density Networks, a statistical machine learning model combining Gaussian mixture models and feed-forward neural networks, to estimate the whole spectrum of resource usage as probability density functions. Modelling the resource usage as a distribution rather than a single point value captures the possible variances caused by resource contentions and interferences from parallel workloads.

6 RESOURCE ADAPTATION

In this review, we refer to resource adaptation in stream processing specifically as horizontal scaling, i.e., adding or removing VMs within the infrastructure to alter the scale of distributed computation. However, there is also vertical scaling that resizes the existing VMs and adjusts the capabilities of hardware in terms of CPU, memory, and network resources. One advantage of vertical scaling is that the underlying resource adaptation remains highly transparent to the application logic, making it easier to leverage resource elasticity at the deployment stage. It can also help reduce the movement of data tuples across the network by consolidating the number of VMs used for data processing and management. Nevertheless, vertical scaling incurs a considerable cost from an operational point of view. The maximal scalability is limited to the capacity of a single host machine, and the adaptation process often involves downtime due to a mandatory system restart. The possible interruption of service makes vertical scaling less preferable when it comes to the cloud deployment of DSPS. The consequence of bringing down the whole streaming system for maintenance can be unacceptable in the presence of continuous inputs and strict latency SLA. Recently, there have been techniques proposed by academia to enact resource adaptation updates while ensuring reliable execution by mitigating the stopping and restarting of the DSPS [142, 164]. But the gap is still huge for these techniques to be mature and for the mainstream cloud provider such as Amazon, Microsoft, and Google to support stop-free vertical scaling in production systems.

In some cases, vertical scaling can be combined with horizontal scaling for right-sizing the VMs for the current load, with a typical use case being workload consolidation [64]. For example, it is profitable to reduce inter-VM latency by consolidating eight medium VMs at 50% load into four medium VMs at 100% load. In addition, vertical scaling can be employed to optimise energy consumption through Dynamic Voltage and Frequency Scaling (DVFS). DVFS is a power management technique that allows processors to dynamically change power states, lowering and raising CPU frequency and voltages on the fly according to the resource demands from virtual machines. It is used by Matteis et al. [35, 37] to explicitly regulate the CPU frequency, by Sun et al. [147] to model the power-to-frequency relationship, and by Shen et al. [139] to turn unused resources into energy savings without affecting application SLA.

While performing resource adaptation in response to the internal and external changes, it is of crucial importance to evaluate the cost-benefit trade-off. The general guideline for deciding whether and how resource adaptation should be triggered is the well-known SASO properties [49]. That is, it exhibits stability (avoids frequent modifications of the current configuration and results in wild oscillation), achieves good accuracy (minimises the number of QoS violations), has short settling time (reaches the desirable configuration quickly), and finally, avoids overshoot (does not overestimate the configuration to meet the needed QoS).

In the rest of this section, we categorise horizontal scaling techniques into two major categories based on how they select the proper scaling time. (1) Proactive approaches that adjust resource provisioning according to the prediction of workload pressure and system behaviour in the future time horizon. (2) Reactive approaches that scale the infrastructure only when necessary as indicated by some threshold breaches or changes of system state.

The choice of proactive or reactive approaches much depends on the predictability of workload pattern and system behaviour. In some cases, the input stream exhibits gradual and repetitive variations in volume and composition, making it possible to learn from the history and apply the obtained knowledge to adjust resource provisioning proactively before the application requirement changes. In other cases, the arriving data stream contains random bursts and drastic workload changes with no clear pattern, leaving the prediction of future system state no longer a viable option [11, 149]. Hence, reactive approaches are required to deal with the bursty load on the best effort basis.

6.1 Proactive Adaptation

Proactive adaptation regards the infrastructure tier as a controllable system requiring certain corrective actions from time to time, e.g., acquiring more resources to tackle under-provisioning or relinquishing over-provisioned resources for cost-efficiency. Therefore, there are continuous controlling loops that monitor the various inputs and outputs of resource management and actively suggest optimal adjustments without delay or overshoot.

A typical workflow of a controlling loop is as follows: (1) The resource estimation module predicts the future system state such as the workload arrival rate and the average input processing latency in the prediction horizon[7]; (2) the system model captures the relationship of various QoS variables, assessing the system's capability to maintain the articulated SLA; (3) the control algorithm solves an optimisation problem to find the best resource allocation for the next loop; (4) perform resource adaptation and adjust the operator parallelism accordingly to avoid data skew and load imbalance.

Based on how the optimisation problem is solved, we generally categorise the proactive adaptation methods into two groups. The first one is *loop-wise control*, which regards each prediction horizon as an independent control interval and derives proactive adjustments by applying the predefined scaling rules to the estimation of the next control loop. Methods falling this group are intuitive and straightforward to implement, but they may suffer from the problem of adjusting for short-term benefits while ignoring the long-term future.

To mitigate this, *Model Predictive Control* (MPC) optimises resource provisioning in a receding prediction horizon that consists of multiple control intervals. At each control interval, the controller solves an optimisation problem to obtain the optimal reconfiguration trajectory over the prediction horizon. However, when it comes to execution, only the first element of the optimal reconfiguration trajectory would be employed to steer the resource adaptation, while the whole trajectory is re-evaluated at the beginning of the next control interval to exploit the updated forecast in the shifted prediction horizon. De Matteis et al. [35–37] employed MPC to achieve QoS-aware and energy-efficient resource adaptation, formulating the optimisation problem as a minimisation of QoS cost, resource cost and adaptation cost. The search space of the optimisation problem is described as a tree structure and the Branch & Bound methods are employed to prune the search tree and reduce the runtime overhead of MPC in a latency-sensitive environment. Meanwhile, Hoseiny Farahabady et al. [63, 64] employed MPC to proactively alleviate the resource contention between collocated applications, in which the optimisation problem is solved by Particle-Swarm Optimization (PSO) with the execution time capped to 1% of the control interval to limit its computational overhead.

6.2 Reactive Adaptation

Based on the metric classification shown in Figure 3, we also categorise different reactive methods by the nature of the triggering metric.

System Metrics Triggered. The system metrics such as CPU utilisation, memory usage, and bandwidth consumption contain raw information on system performance and resource utilisations, thus reflecting the need for adaptation when some metrics have breached certain thresholds. The common problem associated with this type of methods is that the system metrics may not faithfully reflect the application performance. For example, a higher CPU utilisation rate does not necessarily mean higher application throughput and lower processing latency. Instead, it may imply that the current resource provisioning is not sufficient for handling the incoming workload.

[7]Prediction horizon: the period in which the future values of the interested metrics are predicted.

However, methods falling into this category are versatile and easy to implement for being application-agnostic—the simplest example would be monitoring the CPU utilisation at each host, with an upper and lower bound defined to trigger scaling in and out actions [22, 23, 58, 152]. The memory threshold method is also found in Liu et al.'s work [95].

Application Metrics Triggered. The application metrics include not only the application performance perceived by the end-user but also internal metrics from the DSPS that include the service time, the arrival rate, and the length of input/output queue for individual operators.

Lohrmann et al. [104] present a reactive scaling strategy that reacts to latency constraint violations with appropriate scaling actions, which minimises the total resource consumption under a varying load scenario. The same approach is also employed in their Nephele [105] implementation. Xu et al. [164] defined a metric named Effective Throughput Percentage (ETP) for each operator, which captures the state of congestion and estimates the impact of operator output toward the application throughput. The operator with the highest ETP will be given more parallelism and assigned to a new VM for scaling out.

By monitoring the input stream rates and the current processing rates within the DSPS, Cervino et al. [24] detect overload conditions in the operator buffer and then scale the number of used VMs accordingly to maintain the required throughput. Similarly, Vijayakumar et al. [153] defined a derived metric describing the difference between the processing time per data block and the average time interval of receiving one block, so that the adaptation is triggered by the calculated buffer-overflow. Kleiminger et al. [84] monitored the lengths of the input and output queues for stream processors, so that the computation can scale out from an on-premise cluster to clouds when needed. Satzger et al. [134] determined if an operator is overloaded by analysing the length of its incoming message queue, with thresholds hard-coded in the scaling logic to trigger adaptations.

Hybrid Metrics Triggered. Since leveraging system metrics or application metrics alone may not faithfully reflect the actual application performance and resource utilisation, there are some works collecting hybrid metrics to comprehensively trigger reactive resource adaptations. The most common combination is to monitor the operator throughput and the resource utilisation at each host node, to deduce the average processing cost per tuple at the operator granularity. Liu et al. [99] applied this method to trigger reactive resource adaptation, so that the overall application throughput can be maintained at a pre-defined level regardless of the initial allocation of resources. In another work of the same group, scale-in is performed when the input load decreases, and so does the resource consumption of each operator [98]. The scale of adaptation is derived from the monitored load difference and a comprehensive metric of per-tuple processing cost.

Apart from stream processing with a DSPS as middleware, there is a trending serverless and event-driven architecture called Function as a Service (FaaS) that utilises a docker-based runtime to scale up or down automatically in response to demand. It provides a programming model to allow developers to write functional logic, which is completely autonomous and independent of the event sources, to be dynamically scheduled and run in response to associated events from external sources. Such built-in elasticity also means that the resource adaptation is managed by the docker runtime internally and the resource cost is billed by the workload activity rather than per hour of VM utilization. The typical examples of this architecture include Apache OpenWhisk and IBM Cloud Functions, which we will discuss more in Section 11 to shed light on how containers facilitate the resource management of stream processing systems.

7 PARALLELISM CALCULATION

Parallelism calculation answers the question of how many streaming tasks are required for an operator to sustain its assigned workload without causing congestions to the whole application

topology. We have identified two prominent approaches in the literature. The first approach is called performance-driven parallelisation—the resulting parallelism degree is a divisor of the operator input size by the anticipated capacity of each streaming task. The second approach is platform-oriented parallelisation—it first checks the maximum number of parallelism units supported by the provisioned platform and then distributes them as resources among different operators to ensure that the platform is not over-utilised by an excessive amount of processes and threads.

7.1 Performance-driven Parallelisation

In this approach, direct calculation of operator parallelism hinges on the accurate profiling of both operator inputs and the capacity of each streaming task, the latter of which is defined as the maximum number of tuples that a single task can sustainably handle per time unit [99]. There are direct and indirect methods to measure the volume of inputs for an individual operator. The direct methods install a metric collector at the task entrance that automatically gauges the flow traffic and regularly reports to the calculation logic [71], while the indirect method relies on the producer and consumer model to infer the input volume of a particular operator by examining the selectivity[8] of its upstream operators [135]. The state-of-the-art DSPSs are now exposing metrics reporting APIs for light-weight stream monitoring and management,[9] so the hurdle of directly measuring operator inputs has been lowered with the abundance of collected metrics.

In addition to measuring operator input, task profiling is another piece of the puzzle to achieve performance-driven parallelisation. There are a bunch of monitoring and sampling techniques that profile the task performance from different perspectives. The most commonly profiled metrics include the average processing latency per tuple [27, 100], the idleness of task execution [73, 159], and the resource usages of a task entity [98, 99]. The relationship between task capacity and the first two metrics is readily established—a task reaches its maximum capacity when fully occupied with tuple processing under the wall clock time. However, estimating task capacity with the last metric relies on the assumption that this task is hosted by a single thread, which means its peak performance is also limited by the maximum CPU utilisation of a single CPU core.

7.2 Platform-oriented Parallelisation

The rationale of platform-oriented parallelisation is twofold—to avoid over-utilising the available resources with excessive operator parallelism and to help incorporate some rules of thumb suggested by the DSPS developers to make full use of the parallel processing capability. Take Apache Storm as an example, it is suggested that the operator parallelism is a multiple of the number of machines deployed in the platform, and the parallelism of data source is a factor of the number of partitions of the message queue, as such configuration empirically facilitates load balancing between different hosts [100, 145]. As for Apache Flink, the official training guide suggests that using 1 CPU per slot and setting the operator parallelism as a multiple of the number of slots would help achieve balanced slot sharing.[10]

Platform-oriented parallelisation is commonly used in industrial deployment settings as reported by Goetz et al. [51]. Specifically, there is a concept of parallelism unit to describe the parallel processing capability of the platform, which essentially multiplies the number of nodes in the platform by the number of cores available on each node. For instance, there are 160 parallelism units

[8] Selectivity: an operator metric that describes the number of data tuples produced as outputs per tuple consumed in inputs.
[9] Apache Storm: http://storm.apache.org/releases/2.0.0-SNAPSHOT/metrics_v2.html; Apache Flink: https://ci.apache.org/projects/flink/flink-docs-stable/monitoring/metrics.html; Apache Samza: https://samza.apache.org/learn/documentation/0.7.0/container/metrics.html.
[10] https://www.slideshare.net/dataArtisans/apache-flink-training-deployment-operations.

available in a cluster consisting of 10 worker nodes with each incorporating 16 cores. The calculated parallelism units are then regarded as a special type of resources that can be distributed among parallel operators in the topology— the slower the task is in terms of the processing latency, the larger parallelism it gets from the resource pool of parallelism units. They also considered the fact that some tasks may exhibit a higher processing latency because of having intensive communications, so the number of parallelism units can be enlarged 10 to 100 times depending on the number of I/O bound operators present in the topology. This is to ensure that there are enough streaming tasks for the communication-intensive operator to split the workload and perform I/O operations.

8 PARALLELISM ADJUSTMENT

The direct calculation of operator parallelism may not be feasible in some user cases due to the lack of pilot run or monitoring facilities. Also, the results of calculation are prone to profiling errors that adversely affect the system performance. Therefore, an iterative adjustment process is needed to dynamically adapt the parallelism degree in response to the continuous variations of workload and system performance.

8.1 Rule-based Approaches

Rule-based approaches have attracted extensive research attentions due to the simplicity of implementation and effectiveness of adjustments. The core of the method is made of a collection of scaling rules that define the triggering thresholds as well as the corresponding scaling actions. In most cases, the scaling actions are greedy-based, which favour direct mitigation of the threshold violation and converging to suitable parallelism quickly at the expense of optimality. It also means that the resulting parallelism may be trapped in the local optimum and a proper backtrack mechanism is required to search for the global optimum [6].

Rule-based approaches can be generally classified as either static or dynamic in terms of execution.

Static Single Threshold. A static threshold is pre-defined in the scaling logic to trigger parallelism adjustments in a single direction. For example, the threshold on processing latency is one-sided—when the monitored latency exceeds the SLA requirement, the operator parallelism is increased to amortise the processing workload by adding more streaming tasks to the fleet. Besides, Humayoo et al. [65] assessed the necessity of adjustment with a utility threshold to evaluate if the probability of obtaining positive gain outweighs that to incur a loss. Gulisano et al. [52] defined an upper imbalance threshold to ensure the standard deviation of load distribution is below a pre-defined limit. Though setting a single threshold statically makes it fairly easy to implement the adjustment logic, expert knowledge on application characteristics and the platform specification are still required to properly decide the threshold value and the corresponding scaling actions. Furthermore, methods falling into this category lack the ability to scale reversely nor being self-adaptive as the employed threshold is fixed during the complete runtime of the system.

Static Multiple Thresholds. Multiple static thresholds are set in pairs to maintain the concerned parameters within certain upper and lower bounds. For instance, Fernandez et al. [23] defined two thresholds on the average CPU usages of each node to trigger parallelism adjustment from the perspective of local resource utilisation. This approach is also seen in Veen et al.'s work [152]. Kombi et al. [86] divided the estimated amount of operator input by the estimated capacity of a streaming task, where two performance thresholds are defined delimiting a low and a high activity level to trigger the corresponding scaling action. The major challenge for this type of methods is oscillation, where opposite scaling operations are conducted continuously due to the poorly

configured thresholds or overreacting changes [49]. Therefore, a configuration of cooling time is set in practice to conservatively limit the frequency of adjustments and mitigate oscillation.

Dynamic Thresholds. With the knowledge acquired from the evaluation of the previous adjustment results, dynamic thresholds improve the method adaptivity by updating the triggering thresholds and refining the adjustment behaviours at runtime. It also helps mitigate oscillation as the parameters of scaling are dynamically updated with regard to the previous run history. Heinze et al. [58] applied reinforcement learning to reward effective adjustments and punish unnecessary changes caused by inappropriate thresholds. Bilal et al. [10] examined whether a change of parameter value has an overall positive or negative impact on latency and throughput, where the dynamic thresholds are defined as the best performance monitored in the execution history.

8.2 Queueing Theory

The anticipation of operator congestion using queueing theory is not only useful for the estimation and adaptation of resource provisioning but also for deciding the relevant parallelism requirement. Mayer et al. [111, 112] built an adaptive data parallelisation middleware that deduces a stationary distribution of the queue length under a certain parallelisation degree, so that the operator parallelism is adjusted accordingly to make sure that the message buffer's limit is not exceeded with a high probability. Liu et al. [100] employed a queueing network to infer the throughput distribution among operators considering their selectivity and communication pattern, based on which the operator parallelism is scaled in batch ensuring that the capability of the data source and data sink is balanced.

A predictive operator latency model is built on queueing theory and employed by Lohrmann et al. [104] to formulate a linear objective function on the minimisation of total parallelism. They applied a gradient descent search to find the optimal degree of parallelism for each operator that reduces resource footprints while enforcing the latency constraints. Similarly, Fu et al. [46] formulated a latency model based on queueing theory to determine the number of nodes that each operator needs to be placed on; however, their approach is dedicated to computationally intensive applications with no regards to the possible communication overhead and network delays. Cardellini et al. [19, 22] searched for the optimal parallelism by jointly considering operator replication and task placement within an integer linear programming formulation, and this process relies on modelling the underlying computing node as an M/M/1 queue to estimate the response time of a particular operator subject to its parallelism, service rate, and incoming load.

8.3 Control Theory

The versatile control theory also applies to the adjustment of operator parallelism. In Section 6.1, we have discussed various MPC-based algorithms that explore the optimal configuration of the target application under ever-changing operational conditions. The parallelism degree of each operator is part of the configuration, which is updated at the beginning of each control interval [35, 36, 37, 63, 64]. In addition, Gedik et al. [47, 49] investigated the profitability of parallelism adjustment with respect to the changes in workload volume and the availability of resources, where a control algorithm is proposed to manage the operator throughputs and congestion with appropriate parallelism. In Li et al.'s work [92], the operator parallelism is controlled by the comparison of congestion degrees[11] that are measured on the operator's receiving and sending queue, where the strength of intervention could be tweaked by an adjustment coefficient. Floratou et al. [45] presented a throughput-oriented policy that automatically configures the parallelism degree to ensure

[11] The congestion degree for a particular operator queue refers to the ratio of the size of the queued messages to the overall queue buffer size.

satisfactory throughput and alleviate backpressure. Similarly, Stela [164] also relies on monitoring throughput changes to make control decisions—the control algorithm increases the parallelism of the most congested and most influential operator to make full use of the newly added machines during scaling out. In Sun et al.'s work [144, 145], the parallelism degree of each operator is determined in proportion to its computational complexity, which is monitored and measured by the unit of MIPS, i.e., Millions of Instructions Per Second.

8.4 Machine Learning and Game Theory

The adjustment of operator parallelism can also resort to a variety of machine learning techniques. Gaussian processes is employed by Zacheilas et al. [166] to analyse historical data of workload volume and processing latency, so that the parallelism degree can be proactively adjusted to augment the system's performance. By applying incremental learning techniques to different query workloads as training sets, Wang et al. [156] predicted the operator resource usages under several manually supplied candidate configurations. The optimal parallelism is then selected to minimise resource usages while considering the current query requests and stream properties. Game theory is also explored to formulate the elastic parallelism scaling problem as a non-cooperative game, with each operator regarded as an independent agent performing a local control strategy. The operator parallelism is thus determined as the system reaches the agreement of Nash equilibrium [113].

Having introduced a decentralized approach for parallelism adjustment, It is also beneficial to compare it with centralized approaches in terms of flexibility and performance. Generally speaking, centralized approaches have a single component conducting a streamlined decision-making process, which allows for enhanced controllability over various operators and also becomes susceptible to single point of failures. However, decentralized approaches break down the adjustment logic into local control strategies that are run by each operator. The result of adjustment may be less efficient but the robustness of decision-making is improved.

9 SCHEDULING OBJECTIVES

Scheduling plays an important role in successful deployment as it determines how streaming tasks acquire resource allocation and exhibit communication pattern over distributed hosts. It can be considered as a process of trading communication cost against resource utilisation, where the safety of adjustment depends on whether the host machine has enough resources for all the streaming tasks placed on it [3].

While making scheduling decisions, the cost of adjustment must be taken into consideration as moving streaming tasks around can potentially cause severe latency spikes. The oscillating placement of tasks, for example, is one of the most common causes of system instability and SLA breaches, which should be avoided whenever possible to ensure the profitability of scheduling.

The scheduling objective can be multi-fold and not exclusive to a single dimension. In some cases, preferences must be set to sort the competing scheduling targets that cannot be fulfilled at the same time. For example, communication-reduction and load-balancing are two conflicting targets requiring task consolidation and task spreading over distributed resources, respectively. And so are energy-efficiency and fault-tolerance—the former tends to remove any under-utilised components for energy conservation while the latter purposely introduces redundancies to improve reliability. When facing multiple scheduling choices, it is up to the developers to decide which scheduling objective addresses the primary application concern. The one being prioritised is referred to as the primary scheduling objective, and the rest are considered as secondary factors to satisfy particular application requirements. This review mainly focuses on the primary scheduling target, and the taxonomy of task scheduling categorises the existing work based on that.

To evaluate and compare different scheduling policies within the same scope, we have classified the various scheduling objectives into six major categories.

9.1 Fairness-aware Scheduling

The meaning of fairness is twofold when it comes to the scheduling of streaming tasks. First, the amount of workload assigned to each node should be fair, avoiding load-unbalance where part of the computing infrastructure is over-utilised while the other part is under-utilised. This is mainly achieved by scheduling at runtime by placing streaming tasks on different hosts if they tend to experience load spikes at the same time [3]. Second, the resources allocated to each streaming application should be fair, preventing the multi-tenancy mechanism in the mainstream DSPSs from causing application starvation and resource competition. However, it should be noted that being fair in load distribution and resource allocation does not necessarily guarantee a streaming application can meet its SLA requirements [74].

Fairness-aware scheduling is adopted by many open-source DSPSs as their default scheduling strategy. For example, the default scheduler of Apache Storm assigns streaming tasks to computing nodes in a round-robin fashion, achieving coarse-grained load balancing by placing roughly the same number of tasks to each node. The FAIR scheduler of Spark supports the grouping of jobs into pools and setting different scheduling options (e.g., weight) for each pool, ensuring the fairness of resource assignment at different granularities. For instance, the FAIR scheduler can group the jobs by the pertaining user, giving each user an equal share of resources rather than giving each job an equal share. Similarly, the default scheduler of Apache Flink endeavours to make sure that the task slots, each of which run one pipeline of parallel tasks, are utilised in a fair manner.

9.2 Performance-oriented Scheduling

Throughput and latency are the two dominant metrics measuring the performance of a streaming application from the end-user's perspective. Maintaining throughput at the required level is of vital importance to the stability of a streaming system. In a streaming environment, the data sources usually work independently and asynchronously with respect to the other parts of the streaming system. So if the processing facility lags behind in sustaining the required throughputs, the message buffer between the data source and the deployment platform will be overwhelmed by the backlogs, which eventually lead to the system crash [100]. However, the importance of reducing processing latency stems from the fact that streaming applications are latency-sensitive in nature.

Performance-oriented scheduling used to be *platform-centric* in a cluster environment, which aims at producing better performance in a fixed deployment platform by optimising the resource utilisation or reducing the network communication of streaming tasks [4, 27, 41, 43, 72, 120, 144, 163]. However, as cloud computing has enabled dynamic resource provisioning during runtime, performance-oriented scheduling has become *SLA-centric* that focuses on meeting the pre-defined performance targets with elastic scaling on resource and operator parallelism [46, 69, 100].

9.3 Resource-aware Scheduling

Resource-aware scheduling matches the resource demands of streaming tasks to the capacity of distributed nodes, so that the total amount of resources required by the fused streaming tasks can be accommodated by the resources of distributed hosts [3]. Being resource-aware offers the opportunity to consume less computing and network resources to achieve the same performance target [98]. Apache Storm, for example, has a built-in resource-aware scheduler that is derived from [120]. In practice, the resource demands and capacity are described as a multi-dimensional vector, with each element representing a particular resource type, such as CPU, memory, and bandwidth [99, 120]. The scheduling process is thus finding a mapping of tasks to machines such

that the overall resource consumption is minimised and the resource constraints are satisfied. To be more specific, the resource constraints state that the accumulated vector of resource demands requested by the collocated tasks can not exceed the vector of resource availability on that node.

In addition, the need for resource-aware scheduling is driven by the ever-growing use of heterogeneous resources in the streaming infrastructure. The computing nodes could range from energy-constrained mobile devices to powerful virtual machines, which possess different computing powers and connection capabilities. Hence, it is of crucial importance to ensure that the workload assignment does not exceed the node's capacity and the resulting task communications can be sustained by the network facilities connecting to it. Furthermore, the task scheduling on specific hardware such as GPU and FPGA should be optimised accordingly to unlock the potential of the heterogeneous hardware [129, 130].

9.4 Cost-aware Scheduling

Cost-aware scheduling and resource-aware scheduling are strongly related, since they all cut back unnecessary resource consumption for cost saving. However, they also differ from each other as the behaviour of VMs, with their startup time and billing intervals, means that reducing resource usage may not reduce the costs. Cost-aware scheduling has an ultimate goal of minimising the overall monetary cost for hosting the streaming system. In the context of stream processing, the cost optimisation problem is easily complicated by the strict latency requirement, the heterogeneity of resource types, and the diversity of billing models. For example, in a computing cloud with heterogeneous resources, the billing schemes for CPUs, GPUs, and FPGAs can be vastly different and so are the programming efforts that are required to utilise them [53]. The scheduler needs to be aware of the infrastructure, knowing the performance characteristics of different computing nodes while conducting different types of computations, and the characteristics of incoming data to make scheduling plans that reduce the overall costs [130]. Similarly, when the deployment involves multiple geo-distributed data centres, or collaborative Fog, Edge, and IoT networks, the cost of data transmission is non-negligible and must be taken into consideration when making scheduling decisions [28].

9.5 Communication-aware Scheduling

From the perspective of implementation, inter-node communication triggers a cumbersome process involving serialisation, message queueing, and network transmission. In contrast, intra-node communication can be reduced to passing an object's pointer in memory, or being expedited by the use of a concurrent programming framework like Disruptor.[12] As inter-node data transmission incurs much higher resource consumption and significant network latency, it is preferable to place communicating tasks on the same node as long as it does not lead to resource contention. This also implies that communication-aware scheduling is a special type of resource-aware scheduling. But rather than formulating the problem as a bin-packing variant to minimise the overall resource consumption, it has a more specific target of minimising the inter-node communication [4, 27, 41, 43, 44, 73, 163]. For example, SPADE [48] has a optimising compiler that automatically maps the applications to distributed resources, minimising the total inter-node communication while exploiting the available operator parallelism for performance improvements.

To be communication-aware, the scheduler needs to monitor the task communication pattern as well as the resource usage at each computing node. The communication pattern can be represented by a weighted directed graph of streaming tasks, in which the weights associated with vertices denote the task resource requirement and the weights on edges represent the instantaneous

[12] https://lmax-exchange.github.io/disruptor/.

throughput of internal streams or the accumulated volume of data transmission. However, the deployment infrastructure is also regarded as a weighted directed graph of computing nodes, where the weights on vertices denote the node's resource availability and the weights on edges represent the bandwidth capacity of network connection. Therefore, communication-aware scheduling is to find a proper mapping of these two graphs at runtime to minimise the number of messages sent between machines while respecting the constraints on computation and network resources.

9.6 Fault-tolerant Scheduling

Due to the large size of deployment, faults in a stream processing system are not only considered as exceptions but rather normal events. This implies that fault-tolerance should be made a first-class citizen in the scheduling phase to allow fast and efficient error-handling. In a data streaming system, the consequences of faults can range from a single tuple failure to cascading node crashes [66]. A tuple failure affects the timely delivery of messages, which could be caused by the package discarding on overloaded networks. A node crash, however, impairs the proper functioning of stream operators that are allocated to this node. In general, we categorise various fault-tolerance techniques into two groups: (1) state management, which allows stateful operators to survive from possible node crashes, and (2) event tracking, which ensures that messages are delivered with regard to the desired semantic. Schedulers that are fault-tolerance-aware can alleviate the overhead of state management, reduce the risk of event replay, and expedite the recovery process by taking the possible failures into consideration during the placement of streaming tasks [90, 137, 146, 154, 169]. For example, the frequency of state check-pointing can be reasonably decreased by being availability-aware [19]: stateful tasks can be scheduled on more reliable computing nodes while stateless tasks that are fail-fast and easy to recover can be assigned to nodes with relatively lower availability. Also, placing communicating tasks in the vicinity and making sure that the bandwidth of network link is not over-utilised can help reduce the risk of message delivery errors [101].

9.7 Energy-efficient Scheduling

Reducing the total energy consumption is of great interests to the scheduling process [144, 147]. The total energy consumption is unnecessarily increased by the under-utilised computing nodes, so it is preferable to perform workload consolidation periodically to put the low-load nodes into shut-down or low-power mode [98]. Another critical source of energy consumption is the continuous communication among different streaming tasks. Depending on the distance of data transfer as well as the implementation of the underlying network infrastructure, the actual energy consumption of conveying a tuple over a message channel can vary significantly. This implies that the scheduler should also be aware of energy consumptions when deciding the stream routing, putting a large volume of internal streams on wired and reliable network connections rather than channels that are susceptible to interferences to reduce the possibility of retransmission.

10 SCHEDULING METHODS

The previous section covers the various objectives of scheduling but provides little explanation on how these targets can be achieved. In this section, we categorise different scheduling methods into four groups and explain the design and implementation of associated schedulers in detail.

10.1 Heuristic-based Scheduling

The scale of the scheduling problem increases exponentially along with the growing application and platform complexity. Since finding the optimal schedule in such a huge solution space is an NP-complete problem, heuristic methods are preferred over exact algorithms to trade off optimality,

completeness, and accuracy for speed. Aniello et al. [4] pioneered the dynamically scheduling of streaming tasks to improve application performance at runtime, where a greedy heuristic is applied to minimise inter-node traffic and avoid load imbalances among all the nodes. T-Storm [163] extended their work by allowing hot-swapping of scheduling algorithms and fine-grained control over worker node consolidation. The proposed traffic-aware scheduling algorithm has a greedy-based heuristic in its kernel that keeps trying to assign streaming tasks to available nodes with minimum incremental traffic load. Chatzistergiou et al. [27] also proposed an improved heuristic that utilises the domain-specific group-wise communication pattern between streaming tasks to minimise the communication cost, which guarantees to produce a schedule in linear-time outperforming the existing quadratic-time solutions in practical cases. Similarly, Rizou et al. [125, 126] came up with a task placement heuristic to minimise the network load, which is calculated as the bandwidth-delay product of data streams between operators. Sun et al. [147] proposed an energy-efficient heuristic that differentiates the scheduling of critical and non-critical operators to minimise the response time and system fluctuations. R-Storm modelled the scheduling problem as a multi-dimensional Knapsack problem, for which they proposed a heuristic algorithm to put communicating tasks in proximity while ensuring no resource constraints on CPU and memory are violated [120]. The list of heuristic-based schedulers goes on with works done by Cammert et al. [14], Sun et al. [146], and Heinze et al. [56, 57].

It is also worth mentioning that heuristic can play a complementary role alongside the exact algorithms for better execution efficiency. The SODA scheduler [160] for System S, a proprietary DSPS developed at IBM, uses a local search heuristic as a backup solution to the main approach of mixed-integer optimisation. The heuristic method steps in when the CPLEX-based solution fails or becomes too slow to converge. In addition, meta-heuristic has been employed in the scheduling process to improve method adaptivity. Smirnov et al. [143] investigated the use of genetic algorithms to yield throughput improvement as compared to the greedy heuristics, where the task placement is adapted as an evolutionary process utilising the performance statistics gathered at runtime.

10.2 Graph-Partitioning-based Scheduling

As we have discussed in Section 9.5, the scheduling process can be formulated as a graph-partitioning problem where the communication graph is reduced to a set of sub-graphs by partitioning its nodes into mutually exclusive groups. The quality of partitioning is often measured by the total amount of inter-partition communications, the degree of load balance across the platform, and the time required to work out a partition plan. Compared to vanilla heuristics approaches, graph-partitioning-based scheduling takes a different perspective to formulate the problem and is inherently resource-aware and communication-aware. However, it also tends to underestimate the cost of scheduling as the solution is often developed from scratch rather than being gradually optimised from the current placement to achieve the scheduling objective.

By assuming the streaming tasks cannot move after their initial placement, Xing et al. [162] employed a static partitioning method to select an operator placement plan resilient enough to withstand different input rate combinations. For dynamic scheduling, Fischer et al. [43] collected the communication behaviour of applications, built the communication graph at runtime, and then set a partitioning objective function in the METIS software to reduce network loads and balance the CPU usage and bandwidth consumption over the platform. Similarly, Khandekar et al. [81] proposed a minimum-ratio cut subroutine to achieve hierarchical partitioning of the operator graph in System S. Eskandari et al. [41] also discussed hierarchical scheduling of streaming tasks with METIS, proposing a two-phase approach that improves on the traditional k-way partitioning method by allowing to dynamically compute the number of computing nodes required in the

platform. Ghaderi et al. [50] employed a randomised scheduling algorithm with a theoretically provable guarantee on low-complexity, which enables a smooth trade-off between the cost of approaching the optimal partitioning and the queueing performance. In Li et al.'s work [92], the streaming tasks are first partitioned based on the dependency graph of communication, while determining the actual task assignment further involves joint optimisation on the topology structure, inter-node traffic and worker node load-balancing.

The theoretical aspect of graph partitioning in the context of streaming task scheduling has been investigated by Eidenbenz et al. [40]. They proved that optimal partitioning is an NP-hard problem and proposed an approximation algorithm that deterministically achieves a constant-factor approximation under a few consumptions on resource provisioning and processing cost.

10.3 Constraint-Satisfaction-based Scheduling

Constraint satisfaction problems (CSPs) regard the entities of interest as set of objects whose state must satisfy a number of constraints or limitations. Thinking the placement of tasks as objects, task scheduling in stream processing can be naturally considered as a constraint satisfaction problem subject to various resource and SLA constraints and requiring efficient search methods to be solved in a reasonable time. When comparing to the heuristic-based scheduling discussed in Section 10.1, constraint-satisfaction-based scheduling emphasises more on the result optimality and tends to traverse a large area of the solution space to maximise the objective function.

Cardellini et al. [18, 21] formulated an optimal scheduling problem considering the application and resource heterogeneity. The objective function is to minimise migration costs, and the constraints are modelled as the satisfaction of the application SLA. The problem is then solved by CPLEX, a widely used integer programming toolkit. Jiang et al. [74] also formulated a mixed integer program on scheduling to achieve max-min fairness in resource allocation for multiple streaming applications, where the non-convex constraints are converted to several linear constraints using linearisation and reformulation techniques. Schneider et al. [136] proposed a scheduling algorithm for the ordered streaming runtime to minimise synchronisation, global data and access locks, which allows any thread to execute any operator while maintaining the constraints of tuple order in operator communication. Load-balancing is added as an implicit constraint by Zhang et al. [168] to ensure more task assignment will be assigned to the node with the lowest CPU and memory consumptions. For a similar purpose, Liu et al. [102] proposed a runtime-adaptive scheduler that assigns tasks loads in proportion to the processing capacity of nodes. By dynamically migrating tasks assignment from slow nodes to fast nodes, the latency difference between the fastest and slowest nodes is mitigated. Buddhika et al. [13] formulated a resource-constrained problem on scheduling to reduce interference that adversely impacts the performance of streaming computations. They proposed a proactive scheduling algorithm that accounts for the changes in the stream packet arrivals and cluster resource utilisations, which utilises a new data structure of prediction ring to track the amount of workload expected in a given time window.

Constraint satisfaction problems can also be solved by exhausted search. Li et al. [93] trained a model with Support Vector Regression (SVR) on a collection of monitored features to predict metrics like the average latency of tuple processing and the average size of tuple transfer. The resulting scheduler algorithm is essentially an exhaust search algorithm that traverses the whole solution space to find the optimal schedule with the minimised end-to-end latency.

10.4 Decentralised Scheduling

A decentralised scheduler is not a tangible entity that collects global information from the deployment platform and makes holistic scheduling decisions for the whole streaming system. Instead, it offloads the scheduling logic to the individual streaming operator or computing node, regarding

each as an independent agent that collaborates with each other to converge to a feasible scheduling plan. The first prominent benefit of decentralised scheduling is robustness, which eliminates the single point of failure and allows graceful degradation in the presence of computing node crashes—the nodes that are not actively cooperating will be excluded from the scheduling resource pool. The second merit of this design is that it can base the scheduling decision on the accurate prediction of communication latency between different hosts, which is of crucial importance for dealing with streaming systems that are geographically distributed on Edge and Fog cloud.

Specifically, the Vivaldi algorithm [32] —a decentralised approach that has linear complexity with respect to the number of network locations—is often employed to calculate accurate coordinates of distributed nodes in a latency network. Pietzuch et al. [121] pioneered the use of the Vivaldi algorithm to make continuous optimisation in stream processing scheduling without the global knowledge of the system. In their work, a stream-based overlay network is proposed to map the upper streaming system and the underlying physical network, so that the task placement is determined by searching in a multi-dimensional cost space in a decentralised manner. Cardellini et al. [17] presented a distributed and self-adaptive QoS-aware scheduler based on the Vivaldi algorithm, which can deal with infrastructure with non-negligible latencies. Rizou et al. [127] employed the Vivaldi algorithm to form a continuous latency space, and the proposed scheduler ensures that the QoS guarantee on latency is fulfilled while the network load incurred is reduced.

Repantis et al. [124], however, designed a set of fully distributed algorithms to discover and evaluate the reusability of data streams and processing components, enabling sharing-aware component composition while being consistent with QoS requirements. Chaturvedi et al. also studied the reusability of distributed streams in the context of Storm to improve resource efficiency, proposing dataflow reuse algorithms that identify the intersection of reusable tasks and streams to collaboratively reuse the outputs of overlapping dataflow [26]. Zhou et al. [172] proposed a decentralised and asynchronous scheduling algorithm that improves load balancing by dynamically migrating operators from overloaded nodes to lightly loaded ones.

Unless otherwise stated, the schedulers surveyed in the other subsections are centralised designed, which are often collocated on the master node of the deployment platform for the convenience of metric collection and scheduling coordination.

11 GAP ANALYSIS AND FUTURE DIRECTIONS

Although many research efforts have investigated the resource management and scheduling in distributed streaming systems, there exist theoretical and technical gaps to the prospect of an SLA-aware and cost-efficient framework that relieves the deployment burden for application providers. In this section, we discuss the identified gaps and shed light on the future directions on this front.

11.1 Fine-grained Profiling

Accurate profiling of application and system metrics plays an important role in the decision-making process as they reflect the current state of the streaming system and indicate whether the desired SLA requirements have been satisfied. However, most of the existing work based their deployment decisions on coarse-grained metrics such as application throughput, end-to-end latency, operator capacity and the volume of internal streams. These metrics collected at the operator or application level are too general to reveal the actual bottleneck of the data stream, so that the amendments can only be made on a best-effort basis with little guarantee on the adjustment effects. To capture the real culprit that throttles the application performance, a fine-grained profiling mechanism is required to fulfill the following expectations. (1) It should be installed at the task level to obtain fine-grained information such as the lengths of input/output queue, the task capacity on different infrastructure, and the average resource cost for processing a single tuple.

A recent work done by Shukla et al. has explored this idea, proposing a fine-grained profiling approach to collect statistics on the peak input tuple rate supported by the task, as well as the corresponding CPU and memory usage [141]. (2) The application metric collected from the DSPS tier should be cross-validated with the system metrics to identify the probable cause and the severity of the processing bottleneck, allowing accurate amendments to be made in the next adjustment cycle. (3) Proper sampling and quantisation techniques should be employed to reduce the profiling overhead while providing strong enough guarantee on result accuracy.

Other challenges associated with performance evaluation of DSPS include the availability of relevant stream processing workloads and the lack of stable virtual/simulation environments. Yahoo Streaming Benchmark has simulated an advertisement analytics pipeline where the campaign and advertisement data in JSON format are used as workloads [30]. A recent work done by Karimov et al. benchmarked Apache Storm, Apache Spark, and Apache Flink with monitoring data derived from an online video game [78]. But these workloads all fall short on evaluating DSPS at scale for motivated applications domains like IoT, which further requires a fresh design of virtual/simulation environments to reliably measure and retrieve various metric data such as maximum sustainable throughput and resource availability.

11.2 Straggler Mitigation

A straggler is a slow-running entity that adversely impacts the performance of the whole streaming system. It could be a streaming task enduring severe resource contention or data skew or a computing node that is over-utilised or affected by the performance variation of the host cloud. In either case, the local performance degradation caused by the straggler will soon propagate throughout the topology structure due to the the producer and consumer communication model. The first path of propagation is through the operator DAG —with a straggler, the upstream operator will be throttled by the accumulated backlogs, and the downstream operators will stagnate without receiving sufficient inputs. The second path of propagation is through the performance correlation of sibling tasks belonging to the same operator. If one of these tasks becomes a straggler and performs significantly worse than the others, then the logic of tuple emitting will reduce the volume of data stream sent to the other sibling tasks to not overwhelm the straggler. This could lead to under-utilisation on other nodes as the healthy sibling tasks could have been placed in different places processing more inputs.

The straggler mitigation techniques have been initially studied in batch processing systems and then ported to most stream processing systems with micro-batch paradigm. Spark Streaming, for example, has a built-in speculative straggler mitigation technique applicable to various workloads, regardless of being either CPU, disk, or network throttled. This is made possible by having speculative backup copies of slow tasks run in neighbouring nodes. Through extensive evaluation, Khan et al. [80] suggest that using mean/standard deviation instead of median for straggler detection, and that using a confidence level to decide if a task can be executed on a node with a history of abnormal behaviours rather than blacklisting that node entirely.

However, there still lacks enough research attention on detecting and mitigating stragglers for canonical stream processing systems, partially because the one-tuple-at-a-time processing model is more dynamic and less trackable. A straggler mitigation mechanism needs to quickly identify the root cause of the performance deterioration and cuts the chain of propagation with active intervention. A straggling computing node can be detected by its soaring resource usages and slow response time, while a straggler streaming task is revealed by the extended average tuple processing time or the sudden rise in resource consumption. In the context of resource management and scheduling, the possible measures to mitigate stragglers include provisioning new resources, adding more parallelism, or rescheduling the straggler on a different node.

11.3 Transparent State Management

An integrated state management system consists of two parts: (1) State elasticity, which allows dynamically scaling up and down the operator parallelism with a state repartitioning and migration mechanism, supporting the relocation of the operator internal state and providing a guarantee on the semantic correctness during the scaling process. (2) State persistence, which backups the computational states to persistent storage or a different node to mask the loss of states caused by JVM or node crashes. There are some preliminary efforts from both academia and industry toward achieving transparent state management [22, 23, 101, 108]. ChronoStream [161], for example, treats the internal state as a first-class citizen and provides state elasticity to cope with workload fluctuation and dynamic resource allocation. However, significant gaps still present in the following aspects. First, there is limited support for the diverse representation of operator state. In most existing state management frameworks, the abstraction and presentation of operator states are limited to key-value mapping for the ease of implementation. But it is possible that computational states exist in other forms such as graphs, hashes and trees that can hardly be indexed by certain keys. One promising research direction would be supporting arbitrary data structure for operator state representation while keeping the repartitioning and migration process entirely transparent to the end-users. The second gap is to reduce the excessive overhead of state migration, which could be overwhelming if the adaptation of resource provisioning, operator parallelism, and task scheduling have not considered the current state placement. Particularly, there is little research on gradual, stepwise task scheduling that eventually converges to the state satisfying the SLA requirements without incurring too much state migration overhead over a short adjustment period. In contrast, most scheduling algorithms in existence determine a new task mapping from scratch by re-applying the scheduling heuristic, re-invoking a graph partitioning algorithm, or re-conducting an exhausting search in the solution space.

11.4 Resource-availability-aware Scheduling

The existing schedulers have often falsely assumed that, once provisioned, the same amount of resources will be offered to the streaming system throughout its standing lifecycle. Therefore, few of them has considered the fluctuation of node resource availability and what implication it might have for the performance of the streaming system. A notable exception is a recent work done by Cardellini et al. [19], who take the modelling of node and link availability into consideration when studying the problem of optimal operator parallelisation and task placement.

It is common and inevitable to experience fluctuation of resource availability in a distributed cloud environment thanks to two major contributing factors. (1) Multitenancy: multiple tenants of a shared platform may experience performance interference as they compete for limited resources, despite mechanisms like virtualisation and cgroups have provided a certain level of isolation for resource allocation. The temporal and spatial performance variations on Amazon EC2, as reported by Kumbhare et al. [88], can be as severe as 23% of VMs having a normalized core performance worse than 80% of the expectation. (2) Background activities: unexpected background events, such as scheduled system backup, security update, and initialisation of another collocated application could take up a portion of resources that were previously made available to the streaming system. Having a scheduler that is aware of node resource availability can help the streaming system avoid resource contentions.

Resource-availability-aware scheduling is particularly useful when there are no further spare resources for the system to scale out due to the limitation of budget or other performance constraints. In that case, we are interested in changing the mapping of tasks to underlying resources so that the local resource shortage can be amortised over the whole platform. For instance, Imai et al. [68] and Buddhika et al. [13] discussed how to optimize the usage of the available resources

through remapping of tasks without expanding the resource pool. The basic idea is that, if tasks of different operators in the topology process less workload accordingly, their resource consumption is expected to be reduced proportionally. So there is an increasing possibility to find a new task mapping that satisfies the updated resource allocation constraints affected by the fluctuation of availability. Such informed scheduling decision will allow the application performance to degrade gracefully without causing straggler problems discussed in Section 11.2.

11.5 Energy-efficient Scheduling

Apart from reducing the total energy consumption through active workload consolidation, it is also of great interests to cut back the proportion of brown energy consumption through task scheduling.

Over the last several years, the energy supply of the infrastructure of streaming systems has been enriched by the green power generated from renewable sources such as sun, wind, water, and biomass waste. Energy-efficient scheduling intends to reduce the carbon emission and other negative impacts on our environment by scheduling computational-intensive tasks on nodes driven by green power, as well as allocating a large chunk of communication on links powered by green energy. To do this, the scheduler needs to exploit suitable forecast mechanisms to predict the supply of renewable energy in an online fashion, as renewable energy can be intermittent and much more variable than conventional energy from the grid. The scheduler is then committed to produce a task mapping that satisfies the energy supply constraints while trying to maximise the use of green energy. If the DSPS adopts a lambda architecture that span stream and batch processing, then energy-efficient scheduling can postpone the execution of batch jobs, if their deadline permits, until there is enough supply of green energy.

It is also common that saving energy on computation and communication are two conflicting targets that cannot be achieved at the same time through the scheduling of streaming tasks. So a theoretical or empirical model on energy consumption is required to evaluate and compare different scheduling plans, ensuring the overall optimal in the reduction of brown energy consumption.

11.6 Cost Efficiency with Different Pricing Models

The monetary cost of resource usages in clouds largely depends on the actual pricing and billing model chosen by the users. Apart from the on-demand pricing model that has been intensively studied in the literature, a variety of alternative pricing models are also offered by mainstream cloud service provider like Amazon, Google, and Microsoft to help users tailor their choices on resource provisioning and reduce the operational cost. To start with, reserved instances with a fixed-term contract are much cheaper than the on-demand ones, which makes them a good fit to host the baseline workload while leaving the on-demand instances for scaling out when needed. Also, the bidding price model can lower the cost of resource usage significantly as these instances are much cheaper for being hosted on the spare compute capacity in the cloud. However, a streaming system using price-biding instances needs to handle interruptions in infrastructure under a fairly short notice, which imposes great challenges for the latency-sensitive system to adapt task placement and migrate the associated computational state accordingly. A comprehensive resource provisioning and task scheduling model combining the use of on-demand, reserved, and price-bidding resources is a promising research topic that would be welcomed by industry users.

11.7 Container-based Deployment

Containerisation of clouds allows the services and applications to adapt efficiently and operate at an unprecedented scale. Containers offer a logical packaging mechanism that decouples the applications from the environment where they actually run, so there is a clean separation of concerns

by differentiating the procedures of application development and deployment. The ability of containers to run virtually anywhere and the isolation of the CPU, memory, storage, and network resources at the OS-level make it profitable to host streaming applications that are dynamic in nature [12]. The Akka actor runtime,[13] for example, has been extended by Luthra et al. [107] to build a distributed network of Docker containers for easy deployment in the edge-IoT scenario.

However, resource management and scheduling in streaming systems over containers would require an overhaul in the design and implementation of existing DSPSs. The most prominent challenge is transparent state management over the container cloud that is initially designed to host state-less micro-services. The stateful streaming tasks may have to store their computational state externally, which could raise new concerns on the performance of state access. FaaS frameworks, for example, rely on the use of distributed key/value stores for state management. Apache OpenWhisk uses consul[14] as a hierarchical key/value store that is accessible by every component of the system for various purposes such as dynamic configuration, feature coordination, leader election, and so on. Besides, the flexibility of arbitrary placement and dynamic scaling of containers makes it hard to keep track of the destination of each internal stream, so that the tuple emitting logic needs to be revised to make sure that the provisioned containers are coordinated properly in sending and receiving messages.

11.8 Integration of Different DSPSs

The diverse user requirements may require different DSPSs to be deployed at the same time to tackle different use cases. It then raises the questions of how to avoid performance interference between collocated DSPSs and how to select the appropriate middleware that best improves the user experience. There are some preliminary efforts to enable federated execution on top of different streaming engines [39, 94]; however, they all lack the ability to theoretically formulate an engine selection problem for a submitted streaming application, where the objective function and the resource and performance constraints caused by DSPS collocation are clearly defined. It is also interesting to investigate how to concatenate different DSPSs together to host a single streaming application, where each DSPS can handle the part of workload or streaming logic that it excels at processing.

12 SUMMARY

It is of great interest to study resource management and task scheduling in distributed stream processing systems to satisfy the SLA requirements with minimal resource cost. This topic has received extensive research attention in the literature—many have paved the way for SLA-aware, self-adaptive deployment by proposing enabling techniques such as elastic resource scaling, dynamic task scheduling, and runtime operator parallelisation. However, there are still many gaps between the state-of-the-art and the prospect that the monitoring, tuning, and adaptation burden of deployment can be completely offloaded to a comprehensive resource management and scheduling framework, which can address key challenges of dynamic workload characteristics, heterogeneous cloud resources types, and ever-changing SLA requirements without requiring user intervention.

In this article, we summarise the achievements made on this front and identify the gaps to bridge by presenting a comprehensive review of resource management and scheduling techniques in stream processing. Our narrative starts with defining the resource management and task scheduling problem and then organising the research topics of interest around a singular context of

[13] https://akka.io/.
[14] https://www.consul.io/intro/index.html.

achieving SLA-awareness and cost-efficiency while deploying stream processing systems on cloud. We also identified the issues and challenges associated with each research topic and developed a taxonomy of existing work to differentiate the specific work properties and method features. Following the structure of the taxonomy, we discussed each research topic in detail and compared the strengths and weaknesses of different methods that fall into the same category. Finally, we shed light on the promising directions to promote future research in this area.

REFERENCES

[1] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. 2010. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endow.* 3, 1–x2 (2010), 1525–1528. https://dl.acm.org/doi/10.14778/1920841.1921029

[2] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. 2010. S4: Distributed stream computing platform. In *Proceedings of the IEEE International Conference on Data Mining Workshops*. IEEE, 170–177. https://www.cs.cmu.edu/~pavlo/courses/fall2013/static/papers/S4PaperV2.pdf.

[3] Martin Hirzel, Robert Soule, Scott Schneider, Bugra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *Comput. Surveys* 46, 4 (2014), 1–34. https://dl.acm.org/doi/10.1145/2528412

[4] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive online scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. ACM Press, 207–218.

[5] Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. 2010. Lime: A Java-compatible and synthesizable language for heterogeneous architectures. *ACM SIGPLAN Not.* 45, 10 (2010), 89–108.

[6] Nathan Backman, Rodrigo Fonseca, and Ugur Çetintemel. 2012. Managing parallelism for stream processing in the cloud. In *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing (HotCDP'12)*. ACM Press, 1–5.

[7] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. 2013. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. ACM Press, 15–24.

[8] Anne Benoit, Henri Casanova, Veronika Rehn-Sonigo, and Yves Robert. 2009. Resource allocation strategies for constructive in-network stream processing. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 1–8.

[9] Anne Benoit, Henri Casanova, Veronika Rehn-Sonigo, and Yves Robert. 2011. Resource allocation for multiple concurrent in-network stream-processing applications. *Parallel Comput.* 37, 8 (2011), 331–348.

[10] Muhammad Bilal and Marco Canini. 2017. Towards automatic parameter tuning of stream processing systems. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM Press, 189–200.

[11] Ioannis Boutsis and Vana Kalogeraki. 2012. RADAR: Adaptive rate allocation in distributed stream processing systems under bursty workloads. In *Proceedings of the 31st IEEE Symposium on Reliable Distributed Systems*. IEEE, 285–290.

[12] Antonio Brogi, Gabriele Mencagli, Davide Neri, Jacopo Soldani, and Massimo Torquati. 2018. Container-based support for autonomic data stream processing through the fog. In *Proceedings of the 23rd European Conference on Parallel Processing*, Vol. 8374. Springer, 17–28.

[13] Thilina Buddhika, Ryan Stern, Kira Lindburg, Kathleen Ericson, and Shrideep Pallickara. 2017. Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams. *IEEE Trans. Parallel Distrib. Syst.* 28, 12 (2017), 3553–3569.

[14] Michael Cammert, Christoph Heinz, Jurgen Kramer, Bernhard Seeger, Sonny Vaupel, and Udo Wolske. 2007. Flexible multi-threaded scheduling for continuous queries over data streams. In *Proceedings of the 23rd IEEE International Conference on Data Engineering Workshop*. IEEE, 624–633.

[15] Michael Cammert, J. Kramer, B. Seeger, and S. Vaupel. 2008. A cost-based approach to adaptive resource management in data stream systems. *IEEE Trans. Knowl. Data Eng.* 20, 2 (2008), 230–245.

[16] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. 2015. Apache Flink: Unified stream and batch processing in a single engine. *Bull. IEEE Comput. Soc. Tech. Committee Data Eng.* 36, 1 (2015), 28–38.

[17] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2015. Distributed QoS-aware scheduling in Storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-based Systems (DEBS'15)*. ACM Press, 344–347.

[18] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM Press, 69–80.

[19] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2017. Optimal operator replication and placement for distributed stream processing systems. *ACM SIGMETRICS Perform. Eval. Rev.* 44, 4 (2017), 11–22.

[20] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2015. On QoS-aware scheduling of data stream applications over fog computing infrastructures. In *Proceedings of the IEEE Symposium on Computers and Communication*. IEEE, 271–276.

[21] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2017. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurr. Comput.: Pract. Exper.* 43, 34 (2017), 4334–4353.

[22] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. 2016. Elastic stateful stream processing in Storm. In *Proceedings of the International Conference on High-performance Computing & Simulation*. IEEE, 583–590.

[23] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*. ACM Press, 725–736.

[24] Javier Cervino, Evangelia Kalyvianaki, Joaquin Salvachua, and Peter Pietzuch. 2012. Adaptive provisioning of stream processing systems in the cloud. In *Proceedings of the 28th IEEE International Conference on Data Engineering Workshops*. IEEE, 295–301.

[25] Badrish Chandramouli, Jonathan Goldstein, Roger Barga, Mirek Riedewald, and Ivo Santos. 2011. Accurate latency estimation in a distributed event processing system. In *Proceedings of the 27th IEEE International Conference on Data Engineering (ICDE'11)*. IEEE, 255–266.

[26] Shilpa Chaturvedi, Sahil Tyagi, and Yogesh Simmhan. 2017. Collaborative reuse of streaming dataflows in IoT applications. In *Proceedings of the 13th IEEE International Conference on e-Science*. IEEE, 403–412.

[27] Andreas Chatzistergiou and Stratis D. Viglas. 2014. Fast heuristics for near-optimal task allocation in data stream processing over clusters. In *Proceedings of the 23rd ACM International Conference on Information and Knowledge Management (CIKM'14)*. ACM Press, 1579–1588.

[28] Wuhui Chen, Incheon Paik, and Zhenni Li. 2016. Cost-aware streaming workflow allocation on geo-distributed data centers. *IEEE Trans. Comput.* 1 (2016), 1–14.

[29] Zhenhua Chen, Jielong Xu, Jian Tang, Kevin Kwiat, Charles Kamhoua, and Chonggang Wang. 2016. GPU-accelerated high-throughput online stream data processing. *IEEE Trans. Big Data* 3, 99 (2016), 1–12.

[30] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang J. Peng, and Paul Poulosky. 2016. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops*. IEEE, 1789–1792.

[31] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *Comput. Surveys* 44, 3 (2012), 1–62.

[32] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. 2004. Vivaldi: A decentralized network coordinate system. *ACM SIGCOMM Comput. Commun. Rev.* 34, 4 (2004), 15–26.

[33] Wenyun Dai, Longfei Qiu, Ana Wu, and Meikang Qiu. 2016. Cloud infrastructure resource allocation for big data applications. *IEEE Trans. Big Data* 3, 99 (2016), 1–11.

[34] Miyuru Dayarathna and Srinath Perera. 2018. Recent advancements in event processing. *Comput. Surveys* 51, 2 (2018), 1–36.

[35] Tiziano De Matteis and Gabriele Mencagli. 2016. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, 1–12.

[36] Tiziano De Matteis and Gabriele Mencagli. 2017. Elastic scaling for distributed latency-sensitive data stream operators. In *Proceedings of the 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. IEEE, 61–68.

[37] Tiziano De Matteis and Gabriele Mencagli. 2017. Proactive elasticity and energy awareness in data stream processing. *J. Syst. Softw.* 127, C (2017), 302–319.

[38] Marcos Dias de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. 2018. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *J. Netw. Comput. Appl.* 103, 1 (2018), 1–17.

[39] Michael Duller, Jan S. Rellermeyer, Gustavo Alonso, and Nesime Tatbul. 2011. Virtualizing stream processing. In *Proceedings of the 12th International on Middleware Conference*. Springer, 269–288.

[40] Raphael Eidenbenz and Thomas Locher. 2016. Task allocation for distributed stream processing. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications*. IEEE, 1–9.

[41] Leila Eskandari, Zhiyi Huang, and David Eyers. 2016. P-scheduler: Adaptive hierarchical scheduling in Apache Storm. In *Proceedings of the Australasian Computer Science Week Multiconference (ACSW'16)*. ACM Press, 1–10.

[42] Havard Espeland, Paul B. Beskow, Hakon K. Stensland, Preben N. Olsen, Stale Kristoffersen, Carsten Griwodz, and Pal Halvorsen. 2011. P2G: A framework for distributed real-time processing of multimedia data. In *Proceedings of the 40th International Conference on Parallel Processing Workshops*. IEEE, 416–426.

[43] Lorenz Fischer and Abraham Bernstein. 2015. Workload scheduling in distributed stream processors using graph partitioning. In *Proceedings of the IEEE International Conference on Big Data*. IEEE, 124–133.

[44] Lorenz Fischer, Thomas Scharrenbach, and Abraham Bernstein. 2013. Scalable linked data stream processing via network-aware workload scheduling. In *Proceedings of the 9th International Conference on Scalable Semantic Web Knowledge Base Systems*. Springer, 81–96.

[45] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-regulating stream processing in Heron. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1825–1836.

[46] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. 2015. DRS: Dynamic resource scheduling for real-time analytics over fast streams. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*. IEEE, 411–420.

[47] Bugra Gedik. 2014. Partitioning functions for stateful data parallelism in stream processing. *VLDB J.* 23, 4 (2014), 517–539.

[48] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. 2008. SPADE: The system S declarative stream processing engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. ACM Press, 1123–1132.

[49] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic scaling for data stream processing. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1447–1463.

[50] Javad Ghaderi, Sanjay Shakkottai, and R. Srikant. 2016. Scheduling storms and streams in the cloud. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 1, 4 (2016), 1–28.

[51] P. Taylor Goetz and Brian O'Neill. 2014. *Storm Blueprints: Patterns for Distributed Real-time Computation*. Packt Pub. 1–426 pages. Retrieved from https://www.oreilly.com/library/view/storm-blueprints-patterns/9781782168294/.

[52] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. 2012. StreamCloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.* 23, 12 (2012), 2351–2365.

[53] Jiong He, Yao Chen, Tom Z. J. Fu, Xin Long, Marianne Winslett, Liang You, and Zhenjie Zhang. 2018. HaaS: Cloud-based real-time data analytics with heterogeneity-aware scheduling. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems*, Vol. 1. IEEE, 1017–1028.

[54] Thomas Heinze. 2011. Elastic complex event processing. In *Proceedings of the 8th Doctoral Symposium on Middleware (MDS'11)*. ACM Press, 1–6.

[55] Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, and Zbigniew Jerzak. 2014. Cloud-based data stream processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-based Systems (DEBS'14)*. ACM Press, 238–245.

[56] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. 2014. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-based Systems (DEBS'14)*. ACM Press, 13–22.

[57] Thomas Heinze, Yuanzhen Ji, Yinying Pan, Franz Josef Grueneberger, Zbigniew Jerzak, and Christof Fetzer. 2013. Elastic complex event processing under varying query load. In *Proceedings of the 1st International Workshop on Big Dynamic Distributed Data*. Springer, 25–30.

[58] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling techniques for elastic data stream processing. In *Proceedings of the 30th IEEE International Conference on Data Engineering Workshops*. IEEE, 296–302.

[59] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. 2015. Online parameter optimization for elastic data stream processing. In *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC'15)*. ACM Press, 276–287.

[60] Nicolas Hidalgo, Daniel Wladdimiro, and Erika Rosas. 2017. Self-adaptive processing graph with operator fission for elastic stream processing. *J. Syst. Softw.* 127, 1 (2017), 205–216.

[61] Christoph Hochreiner, Stefan Schulte, Schahram Dustdar, and Freddy Lecue. 2015. Elastic stream processing for distributed environments. *IEEE Internet Comput.* 19, 6 (2015), 54–59.

[62] Christoph Hochreiner, Michael Vogler, Stefan Schulte, and Schahram Dustdar. 2016. Elastic stream processing for the Internet of Things. In *Proceedings of the 9th IEEE International Conference on Cloud Computing*. IEEE, 100–107.

[63] M. Reza Hoseiny Farahabady, Hamid R. Dehghani Samani, Yidan Wang, Albert Y. Zomaya, and Zahir Tari. 2016. A QoS-aware controller for Apache Storm. In *Proceedings of the 15th IEEE International Symposium on Network Computing and Applications*. IEEE, 334–342.

[64] Mohammad Reza Hoseiny Farahabady, Albert Y. Zomaya, and Zahir Tari. 2017. QoS- and contention- aware resource provisioning in a stream processing engine. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 137–146.

[65] Mahammad Humayoo, Yanlong Zhai, Yan He, Bingqing Xu, and Chen Wang. 2014. Operator scale out using time utility function in big data stream processing. In *Proceedings of the International Conference on Wireless Algorithms, Systems, and Applications*. Springer, 54–65.

[66] Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. 2012. Deriving a unified fault taxonomy for event-based systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-based Systems (DEBS'12)*. ACM Press, 167–178.

[67] Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. 2013. Elastic stream processing in the cloud. *Wiley Interdisc. Rev.: Data Min. Knowl. Discov.* 3, 5 (2013), 333–345.

[68] Shigeru Imai, Thomas Chestna, and Carlos A. Varela. 2012. Elastic scalable cloud computing using application-level migration. In *Proceedings of the 5th IEEE International Conference on Utility and Cloud Computing*. IEEE, 91–98.

[69] Shigeru Imai, Stacy Patterson, and Carlos A. Varela. 2017. Maximum sustainable throughput prediction for data stream processing over public clouds. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 1–10.

[70] Atsushi Ishii and Toyotaro Suzumura. 2011. Elastic stream computing with clouds. In *Proceedings of the 4th IEEE International Conference on Cloud Computing*. IEEE, 195–202.

[71] Gabriela Jacques-Silva, Bugra Gedik, Rohit Wagle, Kun-Lung Wu, and Vibhore Kumar. 2012. Building user-defined runtime adaptation routines for stream processing applications. *Proc. VLDB Endow.* 5, 12 (2012), 1826–1837.

[72] Jiahua Fan, Haopeng Chen, and Fei Hu. 2015. Adaptive task scheduling in Storm. In *Proceedings of the 4th International Conference on Computer Science and Network Technology*. IEEE, 309–314.

[73] Jiawei Jiang, Zhipeng Zhang, Bin Cui, Yunhai Tong, and Ning Xu. 2017. StroMAX: Partitioning-based scheduler for real-time stream processing system. In *Proceedings of the International Conference on Database Systems for Advanced Applications*, Vol. 3882. Springer, 269–288.

[74] Yuxuan Jiang, Zhe Huang, and Danny H. K. Tsang. 2017. Towards max-min fair resource allocation for stream big data analytics in shared clouds. *IEEE Trans. Big Data* 4, 1 (2017), 130–137.

[75] Supun Kamburugamuve, Leif Christiansen, and Geoffrey Fox. 2015. A framework for real time processing of sensor data in the cloud. *J. Sensors* 2015, 1 (2015), 1–11.

[76] Supun Kamburugamuve and Geoffrey Fox. 2013. Survey of distributed stream processing for large stream sources. *Grids UCS Indiana Edu.* 2 (2013), 1–16.

[77] Supun Kamburugamuve, Karthik Ramasamy, Martin Swany, and Geoffrey Fox. 2017. Low latency stream processing: Apache Heron with Infiniband & Intel Omni-Path. In *Proceedings of the 10th International Conference on Utility and Cloud Computing*. ACM Press, 101–110.

[78] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, RomanSamarev, Henri Heiskanen, and Volker Markl. 2018. Benchmarking distributed stream data processing systems. In *Proceedings of the IEEE 34th International Conference on Data Engineering*. IEEE, 1507–1518.

[79] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (2003), 41–50.

[80] Danish Khan, Kshiteej Mahajan, Rahul Godha, and Yuvraj Patel. 2015. *Empirical Study of Stragglers in Spark SQL and Spark Streaming*. Technical Report. 1–12. Retrieved from http://pages.cs.wisc.edu/.

[81] Rohit Khandekar, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Joel Wolf, Kun-Lung Wu, Henrique Andrade, and Bugra Gedik. 2009. COLA: Optimizing stream processing applications via graph partitioning. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Springer, 308–327.

[82] Alireza Khoshkbarforoushha, Rajiv Ranjan, Raj Gaire, Prem P. Jayaraman, John Hosking, and Ehsan Abbasnejad. 2015. Resource usage estimation of data stream processing workloads in datacenter clouds. arxiv:1501.07020.

[83] Alireza Khoshkbarforoushha, Rajiv Ranjan, and Peter Strazdins. 2016. Resource distribution estimation for data-intensive workloads: Give me my share and no one gets hurt! In *Communications in Computer and Information Science*. Vol. 393. Springer, 228–237.

[84] Wilhelm Kleiminger, Evangelia Kalyvianaki, and Peter Pietzuch. 2011. Balancing load in stream processing with the cloud. In *Proceedings of the 27th IEEE International Conference on Data Engineering Workshops*. IEEE, 16–21.

[85] Boris Koldehofe, Ruben Mayer, Umakishore Ramachandran, Kurt Rothermel, and Marco Völz. 2013. Rollback-recovery without checkpoints in distributed event processing systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS'13)*. ACM, 27–38.

[86] Roland Kotto Kombi, Nicolas Lumineau, and Philippe Lamarre. 2017. A preventive auto-parallelization approach for elastic stream processing. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*. IEEE, 1532–1542.

[87] Sanjeev Kulkarni, Nikunj Bhagat, Masong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter Heron: Stream processing at scale. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM Press, 239–250.

[88] Alok Gautam Kumbhare, Yogesh Simmhan, and Viktor K. Prasanna. 2014. PLAStiCC: Predictive look-ahead scheduling for continuous dataflows on clouds. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 344–353.

[89] Geetika T. Lakshmanan, Ying Li, and Rob Strom. 2008. Placement strategies for Internet-scale data stream systems. *IEEE Internet Computing* 12, 6 (2008), 50–60.

[90] Myungcheol Lee, Miyoung Lee, Sung Jin Hur, and Ikkyun Kim. 2015. Load adaptive distributed stream processing system for explosive stream data. In *Proceedings of the 17th International Conference on Advanced Communication Technology*, Vol. 5. IEEE, 753–757.

[91] Boduo Li, Yanlei Diao, and Prashant Shenoy. 2015. Supporting scalable analytics with latency constraints. *Proc. VLDB Endow.* 8, 11 (2015), 1166–1177.

[92] Chunlin Li, Jing Zhang, and Youlong Luo. 2017. Real-time scheduling based on optimized topology and communication traffic in distributed real-time computation platform of Storm. *J. Netw. Comput. Appl.* 87, 10 (2017), 100–115.

[93] Teng Li, Jian Tang, and Jielong Xu. 2016. Performance modeling and predictive scheduling for distributed stream data processing. *IEEE Trans. Big Data* 7790, 99 (2016), 1–12.

[94] Harold Lim and Shivnath Babu. 2013. Execution and optimization of continuous queries with cyclops. In *Proceedings of the International Conference on Management of Data (SIGMOD'13)*. ACM Press, 1069–1072.

[95] Qian Lin, Beng Chin Ooi, Zhengkui Wang, and Cui Yu. 2015. Scalable distributed stream join processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. ACM Press, 811–825.

[96] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward in-network computation with an in-network cache. *ACM SIGARCH Comput. Architect. News* 45, 1 (2017), 795–809.

[97] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. 2017. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems*. IEEE, 372–382.

[98] Xunyun Liu and Rajkumar Buyya. 2017. D-Storm: Dynamic resource-efficient scheduling of stream processing applications. In *Proceedings of the 23rd International Conference on Parallel and Distributed Systems*. IEEE, 1–8.

[99] Xunyun Liu and Rajkumar Buyya. 2017. Performance-oriented deployment of streaming applications on cloud. *IEEE Trans. Big Data* 14, 8 (2017), 1–14.

[100] Xunyun Liu, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Chenhao Qu, and Rajkumar Buyya. 2017. A stepwise auto-profiling method for performance optimization of streaming applications. *ACM Trans. Auton. Adapt. Syst.* 12, 4 (2017), 1–33.

[101] Xunyun Liu, Aaron Harwood, Shanika Karunasekera, Benjamin Rubinstein, and Rajkumar Buyya. 2017. E-Storm: Replication-based state management in distributed stream processing systems. In *Proceedings of the 46th International Conference on Parallel Processing*. IEEE, 571–580.

[102] Yuan Liu, Xuanhua Shi, and Hai Jin. 2016. Runtime-aware adaptive scheduling in stream processing. *Concurr. Comput.: Pract. Exper.* 28, 14 (2016), 3830–3843.

[103] Giorgia Lodi, Leonardo Aniello, Giuseppe A. Di Luna, and Roberto Baldoni. 2014. An event-based platform for collaborative threats detection and monitoring. *Info. Syst.* 39 (2014), 175–195.

[104] Bjorn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic stream processing with latency guarantees. In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems*. IEEE, 399–410.

[105] Björn Lohrmann, Daniel Warneke, and Odej Kao. 2014. Nephele streaming: Stream processing under QoS constraints at scale. *Cluster Comput.* 17, 1 (2014), 61–78.

[106] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. 2018. Elastic symbiotic scaling of operators and resources in stream processing systems. *IEEE Trans. Parallel Distrib. Syst.* 29, 3 (2018), 572–585.

[107] Manisha Luthra, Boris Koldehofe, Pascal Weisenburger, Guido Salvaneschi, and Raheel Arif. 2018. TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. ACM Press, 136–147.

[108] Kasper Grud Skat Madsen, Philip Thyssen, and Yongluan Zhou. 2014. Integrating fault-tolerance and elasticity in a distributed data stream processing system. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management (SSDBM'14)*. ACM Press, 1–4.

[109] Kasper Grud Skat Madsen, Yongluan Zhou, and Li Su. 2016. Enorm: Efficient window-based computation in large-scale distributed stream processing systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. ACM Press, 37–48.

[110] Lena Mashayekhy, Mahyar Movahed Nejad, Daniel Grosu, Quan Zhang, and Weisong Shi. 2015. Energy-aware scheduling of MapReduce jobs for big data applications. *IEEE Trans. Parallel Distrib. Syst.* 26, 10 (2015), 2720–2733.

[111] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2014. Meeting predictable buffer limits in the parallel execution of event processing operators. In *Proceedings of the IEEE International Conference on Big Data*. IEEE, 402–411.

[112] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2015. Predictable low-latency event detection with parallel complex event processing. *IEEE Internet Things J.* 2, 4 (2015), 274–286.

[113] Gabriele Mencagli. 2016. A game-theoretic approach for elastic distributed data stream processing. *ACM Trans. Auton. Adapt. Syst.* 11, 2 (2016), 1–34.

[114] Jefferson Morales, Erika Rosas, and Nicolas Hidalgo. 2014. Symbiosis: Sharing mobile resources for stream processing. In *Proceedings of the IEEE Symposium on Computers and Communications*. IEEE, 1–6.

[115] Matteo Nardelli. 2016. QoS-aware deployment of data streaming applications over distributed infrastructures. In *Proceedings of the 39th International Convention on Information and Communication Technology, Electronics and Microelectronics*. IEEE, 736–741.

[116] Stephen Neuendorffer and Kees Vissers. 2008. Streaming systems in FPGAs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*. Springer, 147–156.

[117] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. 2017. Samza: Stateful scalable stream processing at LinkedIn. *Proc. VLDB Endow.* 10, 12 (2017), 1634–1645.

[118] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, Kirak Hong, David Lillethun, and Umakishore Ramachandran. 2014. MCEP: A mobility-aware complex event processing system. *ACM Trans. Internet Technol.* 14, 1 (2014), 1–24.

[119] Apostolos Papageorgiou, Ehsan Poormohammady, and Bin Cheng. 2016. Edge-computing-aware deployment of stream processing tasks based on topology-external information: Model, algorithms, and a storm-based prototype. In *Proceedings of the 5th IEEE International Congress on Big Data*. IEEE, 259–266.

[120] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. 2015. R-Storm: Resource-aware scheduling in Storm. In *Proceedings of the 16th Annual Conference on Middleware (Middleware'15)*. ACM Press, 149–161.

[121] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. 2006. Network-aware operator placement for stream-processing systems. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 49–49.

[122] T. Ralf, Muhammad Intizar Ali, Payam Barnaghi, Sorin Ganea, Frieder Ganz, Manfred Haushwirth, Brigitte Kjærgaard, K. Daniel, Alessandra Mileo, Septimiu Nechifor, Amit Sheth, and Vlasios Tsiatsis. 2014. Real time IoT stream processing and large-scale data analytics for smart city applications. In *Proceedings of the European Conference on Networks and Communications*. IEEE, 1–5.

[123] Rajiv Ranjan. 2014. Streaming big data processing in datacenter clouds. *IEEE Cloud Comput.* 1, 1 (2014), 78–83.

[124] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. 2006. Synergy: Sharing-aware component composition for distributed stream processing systems. In *Proceedings of the ACM/IFIP/USENIX International Conference on Middleware*, Vol. 4290. Springer, 322–341.

[125] Stamatia Rizou, Frank Durr, and Kurt Rothermel. 2010. Solving the multi-operator placement problem in large-scale operator networks. In *Proceedings of the 19th International Conference on Computer Communications and Networks*. IEEE, 1–6.

[126] Stamatia Rizou, Frank Durr, and Kurt Rothermel. 2011. Fulfilling end-to-end latency constraints in large-scale streaming environments. In *Proceedings of the IEEE International Performance Computing and Communications Conference*. IEEE, 1–8.

[127] Stamatia Rizou, Frank Durr, Kurt Rothermel, F. Durr, and Kurt Rothermel. 2010. Providing QoS guarantees in large-scale operator networks. In *Proceedings of the 12th IEEE International Conference on High-performance Computing and Communications*. IEEE, 337–345.

[128] Henriette Röger and Ruben Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Survey* 52, 2 (2019), 1–37.

[129] Marek Rychly, Petr Koda, and Pavel Mr. 2014. Scheduling decisions in stream processing on heterogeneous clusters. In *Proceedings of the 8th International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE, 614–619.

[130] Marek Rychlý, Petr Škoda, and Pavel Smrž. 2015. Heterogeneity-aware scheduler for stream processing frameworks. *Int. J. Big Data Intell.* 2, 2 (2015), 70–82.

[131] Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, and Hans-Arno Jacobsen. 2012. Multi-query stream processing on FPGAs. In *Proceedings of the 28th IEEE International Conference on Data Engineering*. IEEE, 1229–1232.

[132] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. 2017. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM Press, 150–156.

[133] Kai-Uwe Sattler and Felix Beier. 2013. Towards elastic stream processing: Patterns and infrastructure. In *Proceedings of the 1st International Workshop on Big Dynamic Distributed Data*. IEEE, 49–54.

[134] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. 2011. Esc: Towards an elastic stream computing platform for the cloud. In *Proceedings of the 4th IEEE International Conference on Cloud Computing*. IEEE, 348–355.

[135] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. 2012. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM Press, 53–64.

[136] Scott Schneider and Kun-Lung Wu. 2017. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 648–661.

[137] Zoe Sebepou and Kostas Magoutis. 2011. CEC: Continuous eventual checkpointing for data stream processing operators. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 145–156.

[138] Vinay Setty, Roman Vitenberg, Gunnar Kreitz, Guido Urdaneta, and Maarten Van Steen. 2014. Cost-effective resource allocation for deploying pub/sub on cloud. In *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*. IEEE, 555–566.

[139] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*. ACM Press, 1–14.

[140] Ce-Kuen Shieh, Sheng-Wei Huang, Li-Da Sun, Ming-Fong Tsai, and Naveen Chilamkurti. 2017. A topology-based scaling mechanism for Apache Storm. *Int. J. Netw. Manage.* 27, 3 (2017), 1933–1952.

[141] Anshu Shukla and Yogesh Simmhan. 2018. Model-driven scheduling for distributed stream processing systems. *J. Parallel Distrib. Comput.* 117 (2018), 98–114.

[142] Anshu Shukla and Yogesh Simmhan. 2018. Toward reliable and rapid elasticity for streaming dataflows on clouds. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems*. IEEE, 1096–1106.

[143] Pavel Smirnov, Mikhail Melnik, and Denis Nasonov. 2017. Performance-aware scheduling of streaming applications using genetic algorithm. *Procedia Comput. Sci.* 108, 6 (2017), 2240–2249.

[144] Dawei Sun and Rui Huang. 2016. A stable online scheduling strategy for real-time stream computing over fluctuating big data streams. *IEEE Access* 4, 1 (2016), 8593–8607.

[145] Dawei Sun, Hongbin Yan, Shang Gao, Xunyun Liu, and Rajkumar Buyya. 2017. Rethinking elastic online scheduling of big data streaming applications over high-velocity continuous data streams. *J. Supercomput.* 74, 2 (2017), 615–636.

[146] Dawei Sun, Guangyan Zhang, Chengwen Wu, Keqin Li, and Weimin Zheng. 2017. Building a fault tolerant framework with deadline guarantee in big data stream computing environments. *J. Comput. Syst. Sci.* 89, 1 (2017), 4–23.

[147] Dawei Sun, Guangyan Zhang, Songlin Yang, Weimin Zheng, Samee U. Khan, and Keqin Li. 2015. Re-stream: Real-time and energy-efficient resource scheduling in big data stream computing environments. *Info. Sci.* 319 (2015), 92–112.

[148] Lauritz Thamsen, Thomas Renner, and Odej Kao. 2016. Continuously improving the resource utilization of iterative parallel dataflows. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems Workshops*. IEEE, 1–6.

[149] Rafael Tolosana-Calasanz, José Ángel Bañares, Congduc Pham, and Omer F. Rana. 2016. Resource management for bursty streams on multi-tenancy cloud environments. *Future Gen. Comput. Syst.* 55 (2016), 444–459.

[150] Ankit Toshniwal, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, and Maosong Fu. 2014. Storm@twitter. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. ACM Press, 147–156.

[151] Jonas Traub, Sebastian Breß, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 2017. Optimized on-demand data streaming from sensor nodes. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM Press, 586–597.

[152] Jan Sipke van der Veen, Bram van der Waaij, Elena Lazovik, Wilco Wijbrandi, and Robert J. Meijer. 2015. Dynamically scaling Apache Storm for the analysis of streaming data. In *Proceedings of the 1st IEEE International Conference on Big Data Computing Service and Applications*. IEEE, 154–161.

[153] Smita Vijayakumar, Qian Zhu, and Gagan Agrawal. 2010. Dynamic resource provisioning for data streaming applications in a cloud environment. In *Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 441–448.

[154] Rohit Wagle, Henrique Andrade, Kirsten Hildrum, Chitra Venkatramani, and Michael Spicer. 2011. Distributed middleware reliability and fault tolerance support in system S. In *Proceedings of the 5th ACM International Conference on Distributed Event-based Systems*. ACM Press, 335–346.

[155] Chunkai Wang, Xiaofeng Meng, Qi Guo, Zujian Weng, and Chen Yang. 2016. OrientStream: A framework for dynamic resource allocation in distributed data stream management systems. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM Press, 2281–2286.

[156] Chunkai Wang, Xiaofeng Meng, Qi Guo, Zujian Weng, and Chen Yang. 2017. Automating characterization deployment in distributed data stream management systems. *IEEE Trans. Knowl. Data Eng.* 29, 12 (2017), 2669–2681.

[157] Di Wang, Elke A. Rundensteiner, Han Wang, and Richard T. Ellison. 2010. Active complex event processing: Applications in real-time health care. *Proc. VLDB Endow.* 3, 1–2 (2010), 1545–1548.

[158] Huayong Wang and Li-Shiuan Peh. 2014. MobiStreams: A reliable distributed stream processing system for mobile devices. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 51–60.

[159] Daniel Warneke and Odej Kao. 2011. Exploiting dynamic resource allocation for efficient parallel data processing in the cloud. *IEEE Trans. Parallel Distrib. Syst.* 22, 6 (2011), 985–997.

[160] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. 2008. SODA: An optimizing scheduler for large-scale stream-based distributed computer systems. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware (Middleware'08)*. Springer, 306–325.

[161] Yingjun Wu and Kian-Lee Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *Proceedings of the 31st IEEE International Conference on Data Engineering*. IEEE, 723–734.

[162] Ying Xing, Jeong-Hyon Hwang, Ugur Çetintemel, and Stanley B Zdonik. 2006. Providing resiliency to load variations in distributed stream processing. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB Endowment, 775–786.

[163] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. 2014. T-Storm: Traffic-aware online scheduling in Storm. In *Proceedings of the 34th IEEE International Conference on Distributed Computing Systems*. IEEE, 535–544.

[164] Le Xu, Boyang Peng, and Indranil Gupta. 2016. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. In *Proceedings of the IEEE International Conference on Cloud Engineering*. IEEE, 22–31.

[165] Lei Yang, Jiannong Cao, Yin Yuan, Tao Li, Andy Han, and Alvin Chan. 2013. A framework for partitioning and execution of data stream applications in mobile cloud computing. *ACM SIGMETRICS Perform. Eval. Rev.* 40, 4 (2013), 23–32.

[166] Nikos Zacheilas, Vana Kalogeraki, Nikolas Zygouras, Nikolaos Panagiotou, and Dimitrios Gunopulos. 2015. Elastic complex event processing exploiting prediction. In *Proceedings of the IEEE International Conference on Big Data*. IEEE, 213–222.

[167] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM Press, 423–438.

[168] Jing Zhang, Chunlin Li, Liye Zhu, and Yanpei Liu. 2016. The real-time scheduling strategy based on traffic and load balancing in Storm. In *Proceedings of the 18th IEEE International Conference on High-performance Computing and Communications*. IEEE, 372–379.

[169] Zhe Zhang, Yu Gu, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. 2010. A hybrid approach to high availability in stream processing systems. In *Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*. IEEE, 138–148.

[170] Xinwei Zhao, Saurabh Garg, Carlos Queiroz, and Rajkumar Buyya. 2017. A taxonomy and survey of stream processing systems. In *Software Architecture for Big Data and the Cloud* (1 ed.). Elsevier, 183–206.

[171] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. 2010. PRESS: PRedictive elastic resource scaling for cloud systems. In *Proceedings of the International Conference on Network and Service Management*. IEEE, 9–16.

[172] Yongluan Zhou, Beng Chin Ooi, Kian-lee Tan, and Ji Wu. 2006. Efficient dynamic operator placement in a locally distributed continuous query system. In *On the Move to Meaningful Internet Systems*. Springer, 54–71.

[173] Qian Zhu and Gagan Agrawal. 2008. Resource allocation for distributed streaming applications. In *Proceedings of the 37th International Conference on Parallel Processing*. IEEE, 414–421.

Received April 2018; revised July 2019; accepted August 2019