A Comprehensive Survey on Parallelization and Elasticity in Stream Processing

HENRIETTE RÖGER, University of Stuttgart - Institute of Parallel and Distributed Systems
RUBEN MAYER, Technical University of Munich

Stream Processing (SP) has evolved as the leading paradigm to process and gain value from the high volume of streaming data produced e.g. in the domain of the Internet of Things. An SP system is a middleware that deploys a network of operators between data sources, such as sensors, and the consuming applications. SP systems typically face intense and highly dynamic data streams. Parallelization and elasticity enables SP systems to process these streams with continuously high quality of service. The current research landscape provides a broad spectrum of methods for parallelization and elasticity in SP. Each method makes specific assumptions and focuses on particular aspects of the problem. However, the literature lacks a comprehensive overview and categorization of the state of the art in SP parallelization and elasticity, which is necessary to consolidate the state of the research and to plan future research directions on this basis. Therefore, in this survey, we study the literature and develop a classification of current methods for both parallelization and elasticity in SP systems.

CCS Concepts: • **General and reference** → *Surveys and overviews*; • **Computer systems organization** → *Distributed architectures*; • **Information systems** → Data management systems;

Additional Key Words and Phrases: Stream Processing, Complex Event Processing, Data Stream Management System, Parallelization, Elasticity

ACM Reference format:
Henriette Röger and Ruben Mayer. 2019. A Comprehensive Survey on Parallelization and Elasticity in Stream Processing. *ACM Comput. Surv.* 1, 1, Article 1 (January 2019), 37 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the surge of the Internet of Things and digitalization in all areas of life, the volume of digital data available raises tremendously. A large share of this data is produced as continuous data streams. Stream Processing (SP) systems have been established as a middleware to process these streams to gain valuable insights from the data. For stepwise processing of the data streams, SP systems span a network of *operators*—the operator graph. To ensure high throughput and low latency with the massive amount of data, SP systems need to parallelize processing. This parallelism comes with two major challenges: First, *how* to parallelize the processing in SP operators. SP systems require mechanism to increase the level of parallelization, which is especially hard for stateful operators that require the state to be partitioned onto different CPU cores of a multi-core server or even different processing nodes in a shared-nothing cloud-based infrastructure. Frequent state synchronization must not hamper parallel processing, while the processing results have to remain consistent. Research proposes different approaches for parallel, stateless and stateful SP. They differ in assumptions about the operator functions and state externalization mechanisms an SP system supports. This led to the development of a broad range of parallelization approaches tackling different problem cases.

Second, how to continuously *adapt* the level of parallelization when the conditions of the SP operators, e.g. the workload or resources available, change at runtime. On the one hand, an SP system always needs enough resources to process the input data streams with a satisfying quality of service (QoS), e.g. latency or throughput. On the other hand, continuous provisioning of computing resources for peak workloads wastes resources at off-peak hours. Thus, an *elastic* SP system scales its resources according to the current need. Cloud computing provides on-demand resources to realize such elasticity [9]. The pay-as-you-go business model of cloud computing allows to cut costs by dynamically adapting the resource reservations to the needs of the SP system. It is challenging to strive the right balance between resource over-provisioning—which is costly, but is robust to workload fluctuations—and on-demand scaling—which is cheap, but is vulnerable to sudden workload peaks. To this end, academia and industry developed elasticity methods. Again, they differ in their optimization objectives and assumptions about the operator parallelization model employed, the target system architecture, state management as well as timing and methodology.

While there are many works that propose methods and solutions for specific parallelization and elasticity problems in SP systems, there is a severe lack of overview, comparison, and classification of these methods. When we investigated these topics in more depth, we found more than 40 papers that propose methods for SP parallelization, and more than 25 papers that propose methods for SP elasticity. An even higher number of papers addresses related problems such as placement, scheduling and migration of SP operators. Every year, dozens of new publications appear in the major conferences and journals of the field. Furthermore, the authors of those papers are located in different sub-communities of the stream processing domain. While they work on the same topics, they have a different view on the problems in SP parallelization and elasticity. Hence, there is an urgent need for a broad investigation, classification and comparison of the state of the art in methods for SP parallelization and elasticity.

1.1 Complementary Surveys

Cugola and Margara [31] presented a general overview of SP systems, languages and concepts, but did not take into account parallelization and elasticity methods. In their survey, Hirzel et al. [67] compared different general concepts to optimize SP operators; however, the authors did not investigate elasticity. Heinze et al. [58] in their tutorial provided a broad overview of SP systems for cloud environments, but did not particularly focus on parallelization and elasticity. Mencagli and De Matteis [36] investigated parallelization patterns for window-based SP operators. They did neither take into account other parallelization strategies nor discussed elasticity methods. Flouris et al. [47] discussed issues in SP systems that are executed in cloud environments. The authors discussed query representations, event selection strategies, probabilistic event streams, eager and lazy detection approaches, optimization with query rewriting, and memory management. However, they only briefly mentioned parallelization and elasticity. Basanta-Val et al. [15] presented patterns to optimize real-time SP systems. These patterns cover operator decomposition and fusion, data parallelization, operator placement, and load shedding. The authors included a theoretical analysis of capabilities and overhead of those patterns. Their focus is on mathematical theory rather than a comprehensive study of specific parallelization and elasticity methods. Many elastic SP systems apply methods from control theory to adapt the parallelization degree of the operators. Shevtsov et al. [129] provide a systematic literature review on control-theoretical software adaptation that goes beyond SP systems. It can be read as a complement of this survey to get a larger view on adaptive software beyond SP systems. In a recent article, Assuncao et al. [33] investigated parallel SP systems with a strong focus on infrastructure and architecture, details on how to implement SP and descriptions of open-source SP frameworks as well as SP in cloud environments. Their discussion of elasticity approaches however lacks a fine grained classification e.g. on parallelization strategy, timing, provided QoS and the methodology focus that is needed to understand the broad range of elasticity concepts. Again, it can be read as a complement of this survey with details how to realize parallel, elastic SP and a stronger focus on frameworks and SP in cloud computing. We conclude that even though surveys for many aspects of SP systems are available, there is a need for a comprehensive study that navigates through available methods for parallelization and elasticity in SP systems to continously deliver high QoS.

1.2 Our Contributions

In this article, we provide a broad analysis of properties relevant for parallel and elastic SP and introduce the methods for parallelization and elasticity. We further summarize and classify the parallelization and elasticity approaches in SP systems. This includes a discussion of research gaps and trends in the field. In addition, we discuss issues that are related to parallelization and elasticity, such as placement, scheduling and migration. These contributions should be helpful both to researchers as well as practitioners to assess the applicability of the state of the art methods to concrete scenarios. Further, they serve as a basis for future research.

1.3 Structure of the Survey

We structured this survey as follows: In Section 2, we discuss fundamentals of parallel SP systems and introduce the main methods for parallelization and elasticity. In Section 3, we classify published work for parallel and elastic SP. Finally, we discuss related topics in Section 4 and conclude the survey in Section 5.

2 GENERAL SYSTEM MODEL AND CLASSIFICATION

In this section, we briefly introduce a generic SP system model, properties of SP systems related to parallelization and parallelization and elasticity techniques.

2.1 General System Model

An SP system processes data streams as the data arrives. It aggregates, filters and analyzes the data items and thus gains fast insights, reactions to observed situations and higher level information. Examples are continuous trend analysis of Twitter feeds, automatic stock trading, fraud detection and traffic monitoring. An SP systems core is the directed, acyclic operator graph that processes and forwards the input data in streams, also called its topology. Also to the topology belong the data sources that emit data items in streams into the graph and the sinks that consume the output. In Fig. 1, we provide a schematic of an exemplary SP system. Data sources are the rectangles on the left, operators the circles, connected by edges that represent the flow of data streams. The rectangle on the right depicts the sink. In a *distributed* SP system, the operators run on multiple processing nodes that are connected via a communication network.

As SP systems are designed to analyze data streams online, *low latency* and *high throughput* are the primary quality of service (*QoS*) goals for SP systems and the major focus of parallelization and elasticity strategies. If a system does not meet its QoS goal, penalty fees may arise or the system,

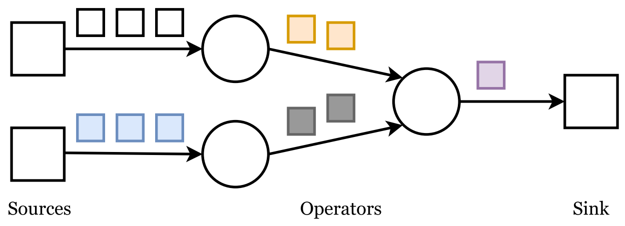

Fig. 1. Schematic of an SP System. Sources emit streams of data items to be processed by a set of operators. A sink consumes the output data stream.

e.g. low latency alarm systems, becomes less useful. Secondary QoS goals, for example balanced load, node utilization, and fault tolerance, are often set to support low latency and high throughput.

2.2 Parallel Stream Processing

If the arrival rate of data items exceeds an operator's processing rate, the operator's input queue grows and induces queuing latency for the data items. Additionally, back-pressure might throttle operators that need to wait until the bottleneck downstream operator processes its input queue. The SP system's QoS goals might be jeopardized. *Parallel SP* reduces queuing latencies and increases throughput as it processes multiple data items simultaneously instead of sequentially. SP systems can parallelize their processing in the multiple ways that we describe in the following. We start with a discussion of those SP system properties that influence the systems parallelization potential and continue with the parallelization methods.

2.3 Properties of Parallel SP Systems

In this section, we introduce those properties of SP systems that influence the SP systems parallelization potential and are necessary for the discussion of approaches in Section 3.1.

2.3.1 Type of SP System. The type of the SP system determines what operations the system supports. We distinguish General SP systems (GP) and the more specialized CEP systems (CEP):

General SP systems. General SP systems apply *continuous operations* on streams of data items. Each operator either produces an output stream of result data items (that itself can be the input stream of another operator) or make the result available to other applications, e.g., by writing them into a data storage or forwarding them to consuming sinks. In general SP systems, we include Data Stream Management Systems (DSMS) that apply *continuous queries* on data streams. A continuous query is a — usually relational— query continuously applied on a changing set of data like a data stream as opposed to traditional database applications that apply a changing set of queries on a fixed data set. Classical examples of non-parallel DSMS systems are Aurora [1] and TelegraphCQ [25]. Modern GP systems are for example Apache Storm [48] and Apache Flink [20].

CEP. CEP systems are SP systems dedicated to detect patterns of events and thereof derive higher level information. The patterns represent a complex *situations of interest*, e.g. the pattern "*Smoke* and *high temperature*" represents *"Fire"*. The input streams of CEP systems consist of *events* triggered by observations of the surrounding world. The operators of CEP systems search for those *sequences of events* in the input streams that fulfill the patterns. If an operator detects a pattern, it emits an output event (sometimes referred to as a *complex event*, e.g. "*Fire*"). When parallelizing CEP systems, it is necessary to divide the input stream onto parallel instances so that patterns will still be detected. Common applications of CEP pattern detection are automatic stock trading [12, 98], financial fraud detection [4, 114] and traffic monitoring [96]. A well-established open source CEP system is, for instance, Esper [41]. Besides that, there are general purpose SP systems that provide CEP functionality as a library, e.g., Apache Flink [20].

2.3.2 Programming Model. The *programming model* defines if programmers use declarative *queries* or an explicit *imperative* implementation to specify the operations or patterns of an SP system. Parallelization solutions for declarative systems might not work for imperative systems and vice versa. In addition, the programming model impacts the data model of the SP system.

Declarative. A declarative query follows syntax and semantics of a specific *query language*. There are different query languages for CEP and for GP systems that differ in their expressiveness, i.e., which kind of queries can be specified in the language. For CEP systems, query languages such as Snoop [24], SASE [145], EPL [41], or TESLA [29], have been proposed. Prominent DSMS-languages are Continuous Query Language (CQL) [8] and (SPL), a query language for IBM's System S [132]. Declarative SP systems automatically deploy an operator graph that implements the query. This automatic deployment step can include optimizations, e.g., fusion or splitting of operators or enabling multiple queries to share operators, c.f. Section 4.3. Declarative SP systems usually require a structured data model for input data items. This data model is defined within the query. The query can then refer to this structure. A common meta model for the structured data model is the relational model.

Imperative. A large group of SP systems follows the imperative programming model. It requires a programmatic specification of the operator graph. This includes both the specification of the operators themselves, e.g., by implementing an API of the SP system, and the specification of the topology of the operator graph, i.e., how the operators are connected. Imperative programming increases expressiveness as the definition of operations is not limited by a declarative language. The widely used open source frameworks Storm, Heron, and Flink follow an imperative programming model [48], [49], [80]. A shortcoming of the imperative model is that due to its less structured nature, automatic optimizations of the operator graph are harder to achieve as in the declarative model. Further, programmatic operators are black boxes to the SP system. The SP system lacks information about the operators internals and cannot exploit them for parallelization. Imperative SP systems give the programmers more freedom for the structure of data items. While these items might still have defined header fields, the payload can be arbitrary.

It is possible to combine imperative and declarative programming. For example, declaratively defined operators are assembled imperatively into an operator graph [133]. The operator graph as the set of operators and their connections is termed the "logical plan" of an SP system. The assignment of this logical plan to a target infrastructure is referred to as the "physical query plan"

2.3.3 Sub-Stream Processing. Operators usually process the input stream sub-stream wise. They extract sub streams based on *keys* or *windows*. *Key-based* extraction groups data items by a the value of a key each data item needs to provide, leading to a sub stream per key value. E.g. in automatic stock trading, a possible key is the stocks class of business, where the trading SP system individually analyzes technology stocks and finance stocks [64].

Window-based extraction builds sub streams according to a *window policy*. A window-policy defines a scope and a slide. The scope describes the window size which, among others, can be count-based (number of data items) or time-based (data items within an interval). The window slide defines the intervals the operator starts a new window on the input stream. It, too, can be time- or count based or rely on a predicate [8, 56, 96]. For a deeper discussion of windows and classification of windowing policies, we point to the literature [50, 70].

2.3.4 *Infrastructure Model.*

The infrastructure on which an SP system is deployed drastically influences the system's performance and thus the need and capability to parallelize the processing. This applies in particular to the type of processing nodes and the memory architecture.

Type of Infrastructure. Infrastructure types for SP systems are single nodes, cluster, cloud or fog solutions. Single node solutions run on a single, multi-core machine with scalability limited by the machine size. An SP system can scale up only, i.e. add more threads, as long as the node size permits. Clusters provide a fixed set of processing nodes. The cluster size limits the SP systems scalability. A common optimization objective for single node and cluster solutions is high resource utilization. SP systems running in a cloud environment have less limited scalability. They face the trade-off between processing performance and cost. The communication between stream sources and the cloud can induce a significant latency which can be critical in low-latency SP applications. Finally, the a fog infrastructure is limited in scalability, too, but often provides low communication latency as the processing can be performed close the sources. Heterogeneous processing nodes might influence the processing speed of the SP application in all types of infrastructure.

Some solutions explicitly integrate specialized hardware, especially GPUs and FPGAs. Exploiting this hardware properly requires additional efforts in the system design. For instance, GPUs provide a high throughput for highly parallel problems, but incorporate a latency and bandwidth penalty for first transferring the input events from the host memory to the device memory.

Memory Architecture. Generally, SP operators communicate asynchronously via message passing. Some systems support operations on shared memory for communication and state management, e.g., when multiple operators are placed on the same host. This reduces communication and state-migration efforts [98] but has higher access synchronization efforts.

2.3.5 *Operator State Models.*

SP systems further differ in their operator state model, i.e. whether and how they support stateful processing and state management in general and per operator. Operator state models are *stateless* or *stateful.* It is common that SP systems comprise stateless *and* stateful operators at the same time. The operator state model influences if, where in the operator graph and to what extend parallel processing is possible. Additionally, it influences the synchronization and access coordination overhead that parallel processing might induce.

Stateless Operators. Stateless operators consider one single data item at a time. It does not store results or information from formerly processing data items; instead, it treats each single data item the same way, regardless of former processing. Examples for stateless operators are filters on temperature data or rescaling of frames in video streams.

Stateful Operators. A *stateful operator* stores received data items or intermediate results as state. It uses and updates this state when it processes subsequent data items. An example is a fire detection application that raises an alarm if the temperature is above 50 degrees Celsius and smoke has been detected in the same area *before.*

The state an operator manages can be limited in scope and lifetime. The scope is limited when an operator processes only the sub streams of a certain set of keys and thus only keeps the respective state for this key set. Time is limited when state is kept for window-based sub streams and dropped once the window is processed. An example for window-limited state is to calculate the average temperature of the last week from daily temperature measurements. It is possible to combine both dimensions, e.g., to limit an operators by a key and additionally limit the lifetime of the state with a window model. A detailed model of partitioned state in SP operators is available [34].

State Management. State management defines if an SP system externalizes state of operators.

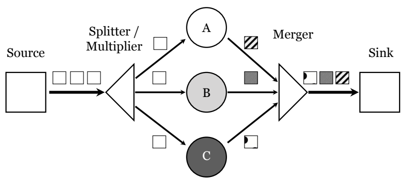

Fig. 2. In task parallelization, an SP system runs different operators in parallel. The splitter or multiplier distributes incoming events to all operators. The merger summarizes the results into one output stream and forwards it to the sink.

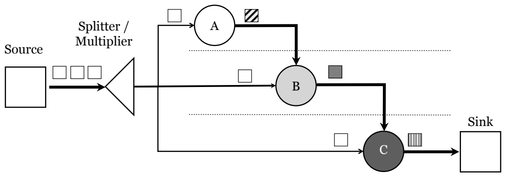

Fig. 3. In pipelining, each operator receives processed the output events its preceding operator. Sometimes, an operator additionally received a copy of the original input stream.

Externalized State. To externalize state, an SP system provides a way to access internal operator state, e.g. via an API or a shared data storage. For instance, in operators with key-partitioned state, internal operator state is externalized to a key-value store [44]. Managed state access become a bottleneck. A discussion of this issue, including a mathematical modeling of the problem, is provided by Wu et al. in [146]. A practical example of applying such a scheme is provided by Hochreiner et al. [69] in their distributed SP system PESP. Finally, Danelutto et al. [32] provide a systematic classification of state access patterns in SP systems.

Internal (Hidden) State. In most SP systems, the internal operator state stays hidden to the rest of the SP system (c.f. Section 3). While this avoids access management, it hampers parallelization, elasticity, load balancing and state migration.

2.4 Operator Parallelization Methods

In the following, we provide a detailed introduction of two parallelization concepts for SP operators: *task parallelization* (Section 2.4.1) and *data parallelization* (Section 2.4.2).

2.4.1 Task Parallelization. In task parallelization, the SP system runs multiple operations on the same input stream in parallel. As an example, see Figure 2: A multiplier replicates the input data items and sends the replicas to operators A, B and C-The merger unifies the resulting data streams into one output stream. An example for task parallelism is the encoding of a video stream in a live-streaming situation into different formats in parallel. Task parallelization enables pipelining where the output of one operator is the input of the next operator. In the context of SP systems, pipelining splits up bigger operators into consecutive sub-operators that can run in parallel (cf. Figure 3 where an operator that detects a sequence “A”, “B”, and “C’ is split into three sub-operators). A requirement for task parallelization is that multiple operations (i.e. tasks) can run in parallel on the same input. Applicability thus depends on the concrete application.

2.4.2 Data Parallelization. The second parallelization method is data parallelization. It executes multiple instances of an operator, i.e., identical operator copies, in parallel on different parts of the input data. The number of instances is the *parallelization degree* of the operator. To enable data parallelization, the input streams needs to be partitionable. A *splitter* component splits the input stream into sub-streams. The splitter can be a process on its own or integrated into operator instances, depending on the splitting strategy. A *merger* aggregates the instances output again into a single stream and, if necessary, ensures in-order delivery to the downstream operators. Figure 4 shows the basic architecture of a data-parallel SP operator.

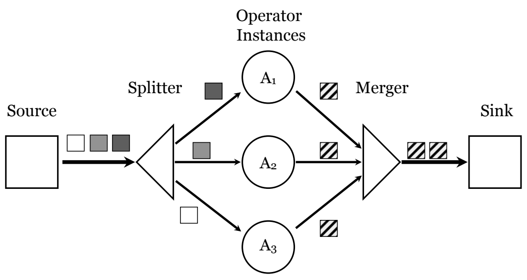

Fig. 4. In data parallelization, a splitter divides the input stream and distributes the data items among instances, i.e., identical copies, of an operator. The instances process in parallel their assigned part of the input stream, i.e., they all perform the same operation on different parts of the data. A merger receives results from each instance and bundles them into one output stream.

Stateful operators require special attention in data parallel SP systems: The input stream should be distributed among the operator instances such that each instance can keep an individual state. This avoids interference in between the different operator instances.

In the following, we describe the three types of splitting strategies common in data parallelization:

Shuffle Grouping. With *shuffle grouping*, the splitter shuffles data items across the operator instances. This is applicable in particular for stateless SP operators (cf. Section 2.3.5), that process data items independent from each other. The splitter itself can be stateless, too, making an independent assignment to an instance for each single data item. A common example is Round-Robin splitting that assigns input data items to the operator instances in a Round-Robin fashion.

If the SP operator implements an associative function, shuffle grouping can also be applied to operators that have key-partitioned state [108]. Each operator instance keeps its own state for each key it has received. A combiner periodically combines the states of each key to the complete state. An example is a word counting application where each instance counts appearances per word and the combiner then calculates the word-wise total. Depending on the number of keys and instances, the combine stage can become very expensive.

A stateful operator can implement shuffle grouping if all instances can access the complete operator state. The order of state access shall unrestricted to avoid sequential processing [32].

Key-based Splitting. Key-based splitting is applicable if operators are stateless or manage state per individual key. Balkesen et al. [14] therefore name it content sensitive splitting as opposed to content insensitive splitting like window-based or shuffle. In the following, we simply denote the value of a key parameter itself as "key". Different ranges of keys are assigned to different operator instances, such that each operator instance keeps the state of a distinct, non-overlapping key range. A schematic of key-based splitting is depicted in Fig. 5. The different gray-scales of the events visualize their key-range, e.g., a specific stock symbol. The splitter forwards data items with the same key-range to the same operator instance.

Window-based Splitting. In *window-based splitting*, the splitter partitions the input event stream into subsequences, i.e windows of data items. It then assigns the windows to the instances of the operator (cf. Fig. 6). Depending on the window policy, Li et al. [87] differentiate between two types of *context*—backward context and forward context—needed in order to determine which windows a given data item belongs to. Backward context of an data item *e* may contain any information about previous data that arrive at the operator, whereas forward context refers to information from

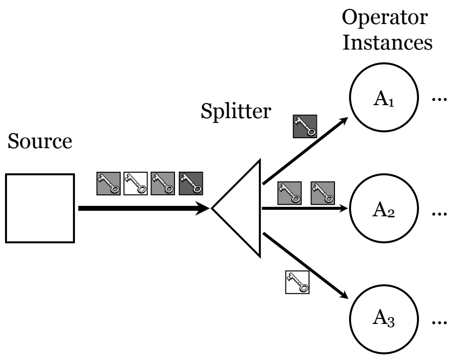

Fig. 5. In key-based splitting, the splitter divides the input stream based on keys. These keys are attributes of the events. Each operator instance is responsible for a sub-range of the total key set.

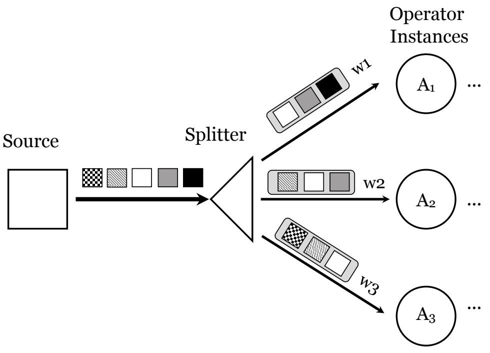

Fig. 6. In window-based splitting, the splitter divides the input stream into windows of subsequent events. Each operator instance processes a sub set of all windows.

subsequent data in the input stream after *e*. In backward context approaches, when an data item *e* is processed in the splitter, a new window can immediately be opened and scheduled to an operator instance, if applicable. However, this is not feasible if the window policy requires forward context.

Pane-based Splitting. We have discussed in Section 2.4.2 that window-based splitting can increase communication overhead for overlapping windows. Furthermore, processing each window from scratch is often not necessary and parts of the computation results in overlapping windows could be shared by all windows. *Pane-based splitting* has been proposed by Balkesen et al. [13] and Li et. al [86]. Pane-based splitting partitions the input stream into non-overlapping sub-sequences, called panes; each pane belongs to one or more windows (cf. Fig. 7). The panes are processed independently and in parallel by operator instances. After the processing, the merger assembles the received results from the panes according to the window policy of the operator. For instance, when the max temperature value in a window with a scope of 1 minute and a shift of 10 seconds shall be computed, the input stream can be split into panes of 10 seconds each. The operator instances compute the max value in each pane in parallel and send the results to the merger. The merger computes the max value of a specific window by determining the max value from all 6 panes that belong to that window.

2.4.3 *Limitations of Operator Parallelization Methods.*

In the following, we briefly discuss each of the presented parallelization methods. We aim to give a better understanding of the opportunities and drawbacks of each method.

Limitations of Task Parallelization. Whilst being a well-established parallelization method, task parallelization, including pipelining, has three major drawbacks: First, it can increase network traffic when each operator has to receive the complete input stream. Second, there is the risk of load imbalance between the operators if they differ in processing speed. Third, the scalability of task parallelization is limited: A given operator can only be divided into a limited set of sub-operators until an atomic operation is reached or, which is more likely, the cost of distributing the processing outweighs the gains drawn from further parallelization.

Limitations of Data Parallelization. The major challenge in data parallelization is to achieve well-balanced workload and in-order delivery of data items to downstream operators. Key-based data

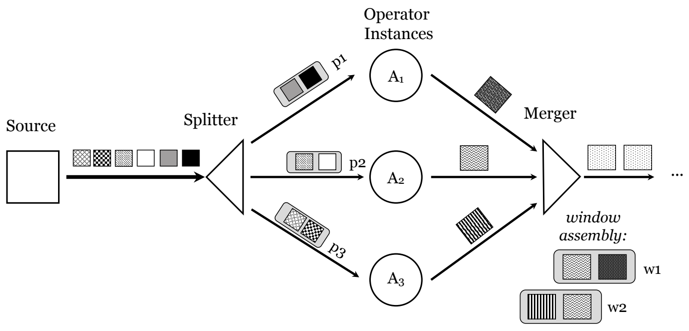

Fig. 7. In pane-based splitting, the splitter divides the input stream into non-overlapping sub-sequences of events, called panes. Each pane belongs to one or more windows. The merger assembles the results according to the windows they belong to.

parallelization has three limitations: Expressiveness, scalability and load balancing. Expressiveness and scalability are limited by the the number of distinct partitions, i.e. distinct key-values. For instance, when checking the stock market for stock patterns, the key-based parallelism is constrained to the number of distinct stock symbols. Further, key-based splitting is applicable only if the data items provide the respective key-values. Explicit load balancing between the operator instances is required when the keys are distributed unevenly across the input data. Situations can occur where one instance experiences a high arrival rate while other instances run idle [118].

Window-based splitting might increase communication overhead: When different overlapping windows are assigned to different operator instances, the data items from the overlap of those windows have to be replicated to all corresponding operator instances. To reduce this overhead, *window batching* assigns multiple subsequent overlapping windows to the same operator instance [13, 36, 99]. The strength of window-based splitting is more flexibility in the input data structure, e.g. does not require keys for splitting.

Pane-based splitting requires the operator function to be dividable into two stages: One stage where the single panes are processed, and one stage where the windows' results are assembled from several panes' results. However, it does not require associative operations as opposed to key-based splitting: the system still processes the data items in sorted groups in the panes, preserving the ordering of the items.

2.5 Operator Elasticity Methods

The employed parallelism of an SP System shall ensure the primary QoS goals (c.f. Section 2.1). As SP Systems are usually long-running, circumstances change: Workload increases or decreases, the hosting infrastructure might have to be shared with other applications and also the content-related properties, e.g. the distribution of keys, can change. For instance, in traffic monitoring, rush hours impose higher workload than low traffic at night. To meet its QoS goals, an SP system has to provide parallelization accordingly, e.g. high parallelism for high workload. Still, a high degree of parallelism requires resources and it is wasteful to *always* provide those resources needed to handle potential workload spikes. Latest with the surge of cloud computing and its pay-as-you-go business model, it becomes attractive to keep the reserved resources as minimal as possible. Thus, running SP systems comes with the continuous task to define the required degree of parallelism to meet its QoS goals while being resource minimal. Therefore, research developed methods to adapt the *degree of parallelism* according to the current circumstances while minimizing resource costs of the SP system. A precise adaptation thereby is the optimal balance between meeting the QoS goals, and minimize the required resources, i.e. the cost. These SP systems, that can adapt the degree of parallelism at runtime, are called *elastic SP systems.*

Elasticity, i.e. adapting parallelism, changes resource requirements of the SP system at runtime, e.g. new processing nodes are required for parallelization increase and instances need to shut down due at a decrease. Therefore, adapting parallelism requires to define how resources can be — fast and efficient — acquired and released, e.g. with idle nodes or anticipated future node requirement and according placement, scheduling and migration strategies. While this survey focuses on parallelization strategies, we mention the former in the discussion in Section 3.2 where applicable and point to related work (Section 4)s for more detailed approaches. Finally, the adaptation overhead needs to be balanced with the adaptations benefits.

In the following, we discuss the properties of elasticity approaches that we use to categorize elastic SP systems in Section 3.2.

2.5.1 Input Data. Elasticity methods base on data from *system information* and *workload information.* System information comprises CPU utilization, throughput, latencies for queuing and processing as well as memory consumption. Workload information is the number of data items in the input streams and their data type. What information is chosen depends mainly on information availability, required precision level and optimization objective. The information can be per individual operator instance, processing node, for a sub-graph or the complete application. Often, the elasticity component uses aggregated data, e.g. averages, predicted trends or a learned distribution.

2.5.2 Timing. The timing when an elasticity method adapts the SP system can be *reactive* or *proactive.* Reactive approaches adapt when the system detects that its QoS goal is violated. In proactive approaches, the system anticipates a violation and adapts *before* it violation occurs, e.g. with prediction models. Reactive approaches adapt the system according to measured data and can thus adapt the parallelization degree to meet the current workload accurately. They are usually simpler because they lack a prediction model. However, due to the delay until the adaptation is fully effective, they might temporarily lead to over-utilization of operator instances, data loss, or SLA violations. Proactive approaches model a future system state and can thus prevent QoS violations. However, the quality of the prediction strongly influences the approaches precision. We also find hybrid approaches that combine reactive and proactive elasticity.

2.5.3 Objective. Each elasticity method has an optimization objective limited by side conditions. The objective aligns with the system's QoS goals, e.g. maximize throughput or minimize latency. Approaches that target high utilization are contrary to approaches that maximize throughput or minimize latency, because high utilization hampers these QoS goals. The side conditions are usually defined by a cost model that describes the SP system's cost, e.g. in terms of used resources. Additionally, adaptation costs can be considered, e.g. if the system halts processing during adaptation which leads to latency spikes. In case of adaptation costs, system stability becomes an important side condition. Some approaches add balanced load, node utilization and fault tolerance to their objective. If noteworthy, we shortly present the cost model when we describe the approaches.

2.5.4 Guarantees. Most elastic approaches keep their QoS goals in a best effort manner, i.e. on average over a wider amount of time. However, some provide real-time guarantees. This includes probabilistic soft-guarantees, e.g. that the application keeps in 90% of the time its latency limit.

2.5.5 Methodology. The methodology defines how an elastic approach comes to its adaptation decisions. Methodology types are *threshold policy driven*, *model driven* and *learning based.* Threshold-policy approaches directly compare the input data against a set threshold. An expressive threshold ensures that elasticity is triggered neither too early nor too late. While thresholds facilitate decision making, finding expressive thresholds, e.g. with profiling, is challenging. Model-driven approaches use the input data to calculate with a mathematical model the new degree. These approaches face the challenge of finding model that represents the system's real behavior. Learned base approaches *learn* a model from measured or profiled data that. These models are usually refined at runtime which improves precision but requires a critical amount of data for learning.

2.5.6 Centralized and Distributed. Elasticity approaches differ in *where* the adaptations are controlled: Many approaches are centralized, i.e. one central component manages the parallelism for the complete operator graph. A centralized component has the global view and can thus thrive towards a global optimum. However, the central component can become a bottleneck and introduce communication overhead. Thus, some solutions implement a distributed approach.

2.5.7 State migration. Changing the parallelization degree of a stateful operator might require state migration. Consider, for example, an application with key-based states where each operator instance is assigned a key range and processes those data items whose key falls into the instance's range. If the parallelization degree, hence the number of instances, changes, so has to the distribution of the key ranges. The respective state of the re-distributed keys needs to be re-distributed accordingly. To manage these state migrations, most elasticity approaches provide a state-migration protocol, e.g., [34, 37]. To realize the migration and avoid inconsistencies, in most SP systems, the processing stops completely or partially during the state migration. Stopping the processing however adds a waiting latency. A common optimization dimension for elastic SP systems is therefore to minimize the number of migrations or the related downtimes [23, 59]. One solution to minimize the number of state-migrations for key-based states is *consistent hashing*. A consistent hashing function ensures balanced load and minimal state migrations when it (re-) distributes key-ranges upon adding and removing nodes [14, 72]. Additionally, Gedik [51] proposes partitioning functions that even outperform consistent hashing.

To enable state migration, many key-based SP systems use the Flux-operator in their splitter [127]. It supports re-partitioning of key-ranges, state migration and load balancing. A buffer absorbs short-term imbalances. For long-term imbalances the buffer cannot absorb, Flux re-distributes the key-ranges to remove the imbalances and consistently migrates the affected state. To keep consistency, the processing at the instances involved in the migration stops and their input is buffered. Due to this buffer, those instances not involved in migration can continue processing.

Shukla and Simman recently proposed two solutions for a reliable and fast *execution* of state migrations [130] for the SP system Storm [48]. Storm natively stops operators and drops input-queues before migration. However, this induces the need for frequent checkpointing and data-item replay to ensure consistent stream processing. The author's first solution stops sources from emitting further data items, processes all input queues before stopping the tasks, checkpoints the state and re-uses it after scaling is finished. This solution avoids re-play of data items and requests checkpointing at migration time only. Their second solution further reduces latencies with faster checkpointing. Additionally, input and ouput queues are stored together with the checkpoints to be resumed by the new instances responsible.

2.5.8 Evaluation. Some elasticity approaches evaluate the effect of an adaptation. They continuously improve their decision making process. Examples are a black-list for non-effective adaptations or model updates of learning agents. Again, we highlight those approaches in our discussion in Section 3.2 where applicable.

3 CATEGORIZATION OF PARALLELIZATION AND ELASTICITY APPROACHES

This section discusses approaches for parallel and elastic SP. Further, two tables give an overview and how the approaches are classified according to the properties introduced in Section 2.2 and Section 2.5, respectively.

3.1 Parallelization

In Table 1, we provide a systematic categorization of the literature on operator parallelization. The first column contains the name of the first author and the reference number in the bibliography. In the “processing" columns, we show the system type and the programming model. As the majority of approaches uses internal state mangement, we highlight in the *External State* column, if a system exposes the state of operators to other parts of the system. We further show the used infrastructure and the memory architecture. For the memory architecture, the default value is *shared nothing*. If shared memory is assumed, we mark this with a tick in the column *SM*. In the parallelization columns, we show the approaches type of parallelization. *TP* is task parallelization, *SG* for shuffle grouping, *KS* for key-based splitting, *WS* for window-based splitting, and *PS* for pane-base splitting. All entries are sorted by the year of publication.

We can see from the table that the amount of work on operator parallelization started to boost around the year 2010, when cloud computing started its surge. Most research targets general SP. The popularity of data parallelization in research papers is very high with a clear dominance of key-based splitting.

In the following, we discuss approaches for SP parallelization in detail. We build three groups: Open source frameworks, parallelization approaches for CEP, and parallelization approaches for GP. Within each group, we highlight the distinctive points in the respective systems or approaches, and provide a finer-grained grouping based on commonalities of the different approaches, if applicable.

3.1.1 Open source frameworks. Many parallelization and elasticity approaches we discuss in this article base their research on an open source SP framework. In this section, we present the most common ones. We explicitly focus on how they enable parallel SP. For each framework, we thus briefly discuss how they provide parallelization options for SP application developers. Only Spark inherently supports elasticity. The other frameworks have interfaces to control parallelism at runtime. For a more detailed discussion of these frameworks with a higher focus on their internal architecture we point to Assuncao et al. [33]. **Esper** [41]: Esper is a declarative SP system for CEP. Developers declaratively define topologies with EPL, a language similar to SQL. The open source version supports scale up and down on a multi-core node with four options for multi threading. Esper does not guarantee in-order processing when multi-threading. The input stream can be split with windows or context. Context can be built from keys, hash values, categories or time e.g. detect events between 9:00 AM and 5:00 PM.

Heron [80]: Heron is a GP SP framework that supports key-based and window-based splitting. It has predefined splitting strategies an additional API for custom strategies. Heron provides multiple APIs to imperatively build SP applications. It is thereby API compatible with Apache Storm. Task parallel processing and pipelining can be incorporated by defining the topology accordingly.

Storm [48]: In terms of parallelization, Storm provides the same options as Heron does. Also input tuples and operator types are the same. Yet alone the API to define topology differs as Heron provides more API-options (e.g. the Streamlet API) than Storm does. Their major differences lie in architectural details that are out of the scope of this article. We point to Kularkni et al[80].

	Processing					Parallelization				
Ref, Name	System Type	Prog. Model	External State	Infrastructure	SM	TP	SG	KS	WS	PS
Esper [41]	CEP	Dec		Cluster	x		x	x	x	
Storm [48]	GP	Imp		Cluster, Cloud, Fog		x	x	x	x	
Heron [80]	GP	Imp		Cluster, Cloud, Fog		x	x	x	x	
Spark [154]	GP	Both		Cluster, Cloud, Fog		x	x	x	x	
Flink [20]	Both	Imp		Cluster, Cloud, Fog		x	x	x	x	
Thies, Gordon [54, 55, 136]	GP	Imp	Stateless	Grid	x	x	x			
Cherniack [26]	GP	Dec		Cluster		x	x	x		x
Brenna [18]	CEP	Dec		Cluster		x		x		
Khandekar [75]	GP	Imp		Cluster (he)		x				
Woods [144]	CEP	Dec	External	FPGA				x		
Neumeyer [110]	GP	Imp		Cluster, Cloud				x		
Andrade [6]	GP	Imp		Cluster				x	x	
Balkesen [13]	GP	Imp		Cluster					x	x
Zeitler [155]	GP	Imp		Cluster				x		
Schneider [124]	GP	Dec		Cluster			x	x		
Gulisano [57]	GP	Dec		Cluster		x		x		
Wu [146]	GP	Dec	External	Cluster	x		x	x		
Hirzel [64]	GP	Dec	External	Cluster				x		
Cugola [30]	CEP	Dec	External	Single Machine	x	x		x		
Fernandez [44]	GP	Imp	External	Cloud				x		
Balkesen [12]	CEP	Dec		Cluster	x	x			x	
Balkesen [14]	GP	Imp		Cluster, Cloud			x	x		x
Wang [140]	CEP	Dec		Cluster		x	x			x
Tang [133]	GP	Imp		Single Machine		x				
Fernandez [45]	GP	Imp	External	Cloud			x	x		
Lohrmann [91]	GP	Imp		Cluster, Cloud		x		x		
Zygouras [158]	CEP	Dec		Cloud				x		
Schneider [125]	GP	Dec		Cloud			x	x		
Rivetti [118]	GP	Imp		Cluster				x		
Mayer [96, 99]	CEP	Imp		Cloud					x	
Wu [147]	GP	Imp	(External)	Cluster				x		
Nasir [108, 109]	GP	Imp		Cluster			x	x		
Saleh [120]	CEP	Dec		Cluster		x		x		
Koliousis [78]	GP	Dec		GPU	x	x				x
Zacheilas [153]	CEP	Dec		Cloud				x		
Nakamura [107]	GP	Imp		Fog		x				
Gedik [53]	GP	Dec		Single Machine	x	x		x		
Mayer [97]	GP	Imp		Cloud				x	x	
Rivetti [117]	GP	Imp	Stateless	Cluster			x			
Schneider [126]	GP	Dec	Stateless	Cluster (he)			x	x		
Katsipoulakis [73]	GP	Both		Single Machine, Cluster	(x)		x	x		
Mayer [98]	CEP	Imp	External	Single Machine	x				x	
Mencagli [105]	GP	Imp		Single Machine	x				x	
Mencagli [103, 104]	GP	Imp		Single Machine	x					x

Table 1. Categorization of SP operator parallelization literature. The "Processing" Columns show the type of SP system. “State External" marks approaches that either externalize state or are limited to stateless operators. The Infrastructure section shows the target infrastructure and if the approach uses shared memory (SM). The last columns mark Task Parallelization (TP), Data parallelization with Shuffle Grouping (SG), splitting key-(KS), window-(WS) or pane-based (PS). Open source frameworks are followed by research approaches in order of their date of publication.

Spark Streaming and Structured Streaming by Spark[154]: As Spark applications originally processed batches, the Spark Streaming extensions process streamed data in micro batches. *Structured Streaming* interprets data streams as an unbounded table where each new data item extends the table. Operators are queries on this table. A specific schema for input data is required. Data can be split using properties (i.e. keys) and aggregated as windows. Operators are defined declaratively. The parallelism degree can be set explicitly in the topology or via a default value in the configuration files. Opposed to the other frameworks, Spark has a Dynamic Resource Allocation module for elasticity. According to the documentation, it releases unused resources in low workload times and requests them again in peak workload times.

Flink[20]: Flink supports key-based and window-based splitting. The parallelization degree for data parallelism can be set in the topology implementation or via an interface. As default, Flink places one instance per operator on each core. To the best of our knowledge, custom grouping mechanisms are not available. Besides a DataStream API, Apache Flink provides two APIs - the Table API and SQL- API - to use relational-oriented query descriptions. For CEP-oriented queries, a CEP library is available.

3.1.2 Parallelization in CEP.

This section summarizes approaches for parallelization in CEP.

Brenna et al. [18] extend Cayuga [39], a centralized, multi-query SP system for pattern matching. They enable pipelining and key-based data parallelization and distribute the processing of Cayuga. It resembles the pipelining approach of Balkesen et al. [12] (cf. Section 2.4.1).

Woods et al. [144] implement the CEP operators directly on FPGAs (Field Programmable Gate Arrays) for fast CEP. They focus on the so called "network-memory-bottleneck". This bottleneck occurs when a SP system writes the content of the event network packages to the main memory to be accessible by the CPU. The FPGA interprets the event network packages without writing them to main memory first. Only detected patterns are forwarded to the main memory. To exploit the parallelism of FPGAs, the authors introduce a query language that uses partition keys and predicates. The system keeps state per key. As FPGA hardware limits the number of keys to be stored, the authors propose a time limit for the storage of keys. With this limit they discard the state of a key, if out of a fixed number of past events in the input stream, none has had the specific key. While the approach is less flexible and scalable due to its hardware-centric nature, it can be beneficial for highly latency sensitive applications.

Cugola and Margara [30] propose two algorithms for pattern matching in CEP operators. The first algorithm, "automata-based incremental processing (AIP)", detects patterns with a non-deterministic finite automaton, while the second algorithm, "column-based delayed processing (CDP)", uses lazy pattern matching. Both algorithms enable pipelining parallelism in multi-stage pattern matching operators, and task parallelism when multiple queries evaluate the same input event stream. An outstanding merit in the work of Cugola and Margara is that they take into account a heterogeneous computing environment with CPUs and GPUs. Based on evaluations, the authors recommend to use the GPU if there are few, but complex operators in the system, and multi-core CPUs for a high number of less computational-intensive, operators.

Balkesen at al. [12] propose the window-based data parallelization approach "Run-based Intra-operator Parallelization (RIP)" for CEP. They implement the operator query as finite state machines (FSM) and start a new FSM instance for each incoming event. Additionally, each event is processed by the already running FSM instances. The authors propose to split the input event stream into overlapping batches of events based on the window policy of the operator. Each batch is assigned a thread that then runs those FSM instances started by the events in that batch. This approach requires bounded window sizes.

The system proposed by Mayer et al. [96] is a distributed, window-based data parallelization framework. Input event streams of an operator are split into windows that are scheduled to an elastic set of distributed operator instances, where each operator instance processes its assigned windows "from scratch". This leads to a very high expressiveness of the framework, as the authors show on operators defined in the Snoop [24] and CQL [8] query languages; basically, any window-based operator can be integrated into the framework. To expose the window policies of the operator to the splitter, a programming API has been integrated into the system, so that operators can be plugged into the system with minimal inference to the operator code.

Later work of Mayer et al. [99] discusses the trade-off between communication overhead and load balancing when performing window batching. The authors propose a model-based batch scheduling controller that predicts the latency peak when assigning a window to an operator instance. Based on that, the controller assigns the windows to the operator instances in such a way that communication overhead is minimized while a latency bound in the operator instances is met.

Wang et al. present in [140] an approach that computes sequence pattern matches by splitting the event stream into uniformly sized batches, which resemble panes. Operator instances process those panes in parallel. The pattern matching is thus done per pane at first. Further, the relationships between events that build any sub-pattern of the complete pattern are stored. A merge stage finds complete patterns by stitching together the sub-patterns reported from the operator instances.

Hirzel [64] proposes a pattern syntax and a translation scheme for IBM's System S that supports pattern matching. An algorithm translates the match-expressions (the pattern to be found) into nondeterministic finite automata (NFAs) at compile time and generates C++ code for it. A "partition map" stores each NFA-state together with the aggregated results computed so far for the state. A "partition-by" attribute in the match-expression selects all states from the map that are relevant when a new event arrives. Parallelization is achieved by splitting the input stream on the same keys (partition-by attributes) that are used to select the partition maps.

Zygouras et al. [158] implemented a highly distributed and parallel Big Data processing system that includes stream and batch processing capabilities. Special about this approach is that it combines the frameworks Esper [41] and Storm [48].

Saleh et al. [120] consider stream and operator splitting of CEP applications. As a basis, the authors use PipeFlow, their distributed CEP system that provides features for data parallelization [121]. The input stream can be split based on keys. However, task parallelization with operator splitting is the focus of their work. The authors partition each CEP operator into sub-operators by rewriting the query according to rewriting rules. The rewritten query is then easier to split. To decide about the size of partitions, the system statically assigns costs for latency and memory consumption for each possible partition. A greedy algorithm then finds the cost-optimal set of operator graph partitions for the available set of computing nodes.

Mayer et al. [97] developed GraphCEP, an SP system that allows for combining stream analytics with graph processing. Event streams are partitioned by a key and fed into operator instances. Each operator instance has access to a graph processing engine [95] to perform heavy-weight parallel computations on a large, shared graph. Further, the merger allows for stateful computations that combine results from multiple operator instances.

In a window-based splitting approach, an event may belong to multiple windows due to a window overlap. *Consumption policies* can define, that if an event contributes to one detected pattern, it cannot be part of another pattern detection, possibly in another window [157]. Hence, the processing of overlapping windows becomes interdependent. Some SP pattern definition languages that enable these policies are Snoop [24], Amit [2] and TESLA [29]. The SPECTRE system by Mayer et al. [98] is the first work that addresses this interdependency by means of speculative processing. In particular, SPECTRE creates multiple versions of dependent windows that assume different event consumptions and assigns the most probable versions to operator instances. SPECTRE is designed for multi-core shared memory environments.

3.1.3 Parallelization for General Stream Processing. The following paragraphs discuss parallelization in general SP systems. Due to the high amount of approaches, we further group them according to the parallelization type, i.e. task or data parallelization. A special case are key-based data parallelization techniques: This biggest group of approaches is further split up for approaches that provide advanced key-partitioning functions, approaches that apply key-based splitting and approaches that focus on key-based state management. Please notice that these groups overlap and we assigned approaches according to the approaches focus.

Approaches implementing Task Parallelization. The StreamIt language by Thies et al. [136] is a high-level programming language that supports both task parallelization and pipelining. StreamIt provides a programming abstraction for managing the event streams and operators. The compiler of StreamIt is described in two further publications [54, 55], leveraging and optimizing task, pipeline and data parallelism.

COLA by Khandekar et al. [75] is a pipelining optimization algorithm for IBM's System S. It optimizes queries at compile time, before the deployment of the operator graph. It fuses operators from the logical plan of the query into coarser-grained operators. The fused operators process the input stream in a pipelined way. The fusion balances communication cost and CPU capacities of the heterogeneous processing nodes.

Lohrmann et al. [91] propose Nephele-Streaming that extends their batch processing framework Nephele [141]. To enable SP on top of Nephele, Lohrmann et al. adopt a *micro batching* approach [154]. It processes data items in a stream of small batches. By changing the granularity of the operator graph, Nephele-Streaming enables pipeline parallelism in the operators.

Nakamura et al. present in [107] an SP middleware that is based on their earlier system called "Information Flow of Things" (IFoT) [151]. They designed a layer fog-computing architecture that uses Raspberry Pi-nodes. On these nodes run so called neurons that perform real-time data stream analysis. The middleware divides each application into tasks according to recipes that describe how the input data should be processed within the application. Tasks might be shared by different applications. Each neuron executes a set of tasks and can exchange information with other nodes via the connected "neuron layer". The input and output event stream of the sensors and actuators is managed by the neurons using the publish/subscribe paradigm [42].

Tang and Gedik [133] introduce task parallelism and pipelining in an SP system. In their system, each threads processes a sequence of operators as a sequence of function calls. To enable pipelining, an additional thread inserted into this sequence to execute the downstream operators and frees the upstream thread to process the next tuple. Their approach provides an elasticity mechanism we describe in Section 3.2. It runs on a single, multi-core node.

Approaches implementing Shuffle Grouping. Approaches that implement shuffle grouping focus on load balancing. Schneider et al. [126] propose an approach that balances load for parallelized stateless operators where the operator instances may differ in throughput. Their System S extension monitors the TCP blocking rate per connection between the splitter and each operator instance. Minimizing the maximal blocking rate among all operator instances balances the load.

Rivetti et al. [117] propose an approach that tackles imbalanced processing latencies of instances. They focus on applications where the data item processing latency can depend on the content and propose "Online Shuffle Grouping", an algorithm for proactive online scheduling of input data based on an estimation of the data items processing time.

Another splitting variant mixing key-based splitting and shuffle grouping is known as "partial key grouping", proposed by Nasir et al. [108]. The shuffling phase is restricted to two deterministic options according to two different hash functions (i.e., key-based splitting with two hash functions instead of one). A number of parallel splitters can pick dynamically to which of the two operator instances to send an event, focusing on the one with less load. This optimization technique is known as "the power of two choices" and provides significant improvement in load balancing [10]. Later, Nasir et al. extend their approach to allow for more than two choices for "hot" keys that impose most of the workload [109]. In all of these approaches, a combiner is needed to combine the shuffled state of each key.

An early attempt of key-aware shuffle grouping has been proposed by Balkesen et al. [14]. When an SP operator has key-partitioned state, the input stream is split by a variant of consistent hashing ("frequency-aware hash-based partitioning"). This variant considers the key frequencies. In particular, the k keys with the highest frequency are further split into sub-keys that are shuffled around the operator instances—the number of splits depends on the frequency.

Recently, Katsipoulakis et al. [73] proposed to consider aggregation cost in the combiner when deciding where to route which key. Based on a mathematical formulation of imbalance *and* aggregation cost (where aggregation cost directly depends on the number of operator instances that receive and process events with the *same* key), they propose several heuristic splitting methods that aim to minimize both imbalance and aggregation cost for a given operator and workload.

In a data-parallel SP operator, the splitter can become a bottleneck if the input event rates are very high. Zeitler and Risch [155] address this problem by proposing a two-stage splitting processing. In the first stage, they split the input stream into fixed-size batches. They then route these batches to a number of parallel splitter instances. Those parallel splitters then perform the actual key-based event routing to the operator instances. The authors provide a mathematical model to compute the optimal batch size for the first splitting stage as well as the optimal number of parallel splitters. The approach works both for stateless splitters as well as key-based splitters.

Key-partitioning Functions. This section summarizes SP approaches apply key-based splitting and put a particular focus on the key-partitioning function.

Rivetti et al. [118] propose an algorithm that learns which keys are currently the most common ones. Based on that, they apply a greedy algorithm to compute a nearly load optimal distribution, balancing frequent and rare keys among the available nodes. The same problem is also tackled by Zacheilas et al. [153], who formulate it as a variation of the job shop scheduling problem and apply an extension of the "Longest Processing Time" algorithm, a greedy heuristic solution.

In [26], Cherniack et al. deploy the SP system Aurora [1] in a distributed setting. To enable data parallelism, they split the operator ("box splitting") and add a splitter and merger instance. The splitter (called "filter with a predicate") routes tuples based on predicates, giving the user advanced options to specify the splitting key with user-defined predicates. The merger can implement, e.g., a union of the output tuples or a sorting algorithm. The authors provide a deeper discussion on challenges of splitting and merging data streams.

Gulisano et al. [57] propose "StreamCloud", an elastic and scalable SP system. Their key-based data parallelization method minimizes distribution overhead. They propose partitioning functions for different stateful SP operators (equijoin, cartesian product, and aggregate operator).

State Management for Key-based Splitting. Fernandez et al. [44] propose SEEP, a key-based data-parallel SP system. In SEEP, operators expose their internal, key-partitioned state to the SP system through a set of state management primitives. SEEP performs state management both for scale-out as well as recovery of operators. In their later work, Fernandez et al. [45] propose the abstraction “stateful data-flow graphs” (SDG) that separates data from mutable operator state. In their model, state elements can be partitioned by a key and dispatching is performed by hash- or range-partitioning on a key or by shuffle grouping. If a state element cannot be partitioned, it is replicated, and state updates are combined by a “merge logic”. Further, Fernandez et al. provide a tool “java2sdg” [43] that translates annotated Java programs to SDGs for execution in the SDG runtime system.

Chronostream by Wu and Tan [147] splits the computational state (based on keys) into so-called slices. This state splitting supports horizontal and vertical scaling. The slices are distributed and replicated across different machines, leading to efficient load balancing and enabling fault tolerance.

Wu et al. [146] extend IBMs System S and enable it to handle shared state in data parallel processing. They implement a round-robin and a hash-based splitting routine. Special about their approach is a theoretical model. It analyzes if parallel processing improves the performance of an SP systems that employs shared state. The model predicts the waiting time and the time overhead that the shared state access induces.

Approaches implementing Key-based Splitting. Schneider et al. [124] present a compiler and a runtime environment for their SP language SPL [65] (Stream Processing Language) that enables data parallelization. Based on hints provided in SPL, the complier defines parallel regions, which are sequences of multiple operators and automatically replicates them. The system supports stateful and stateless operators and splits the input stream of a parallel region based on keys or, if the operator is stateless, with Round-Robin shuffle grouping. Fusion of operators further reduces network communication. In a later publication [125], the authors add support for operators that produce a dynamic number of output data items per input data item.

Gedik et al. [53] propose an approach for “pipelined fission”, a combination of pipelining and key-based data parallelization. They fuse groups of operators into so-called pipelines with a heuristic optimization algorithm. Different threads can then execute these pipelines independently to enable parallelism. Additionally, stateless or key-partitioning pipelines can be further parallelized by a split–process–merge architecture to exploit data parallelism.

Neumeyer et al. [110] propose the highly scalable SP system S4. The system scales with Processing Engines (PE) where one PE is responsible for a specific key and processes all corresponding data items. A processing node (PN) manages multiple PEs, i.e., a subset of the key domain, and instantiates new PEs if a new key occurs that is not covered by an existing PE yet.

Approaches implementing Window-based Splitting. Andrade et al. [6] present a low-latency SP implementation in IBM’s System S. This implementation splits the input stream of an operator on two levels: On the first level key-based, on the second level, for each of the key ranges, window-based. Thus, it is for example possible to calculate an average over a time window, i.e. a window-based operation, for the stock value of different companies, i.e. on a key-based split input data stream. The authors name this pattern the “split/aggregate/join" pattern.

The work of Mencagli et al. [105] propose a novel execution model, “agnostic worker” for multi-core shared memory environments. An agnostic worker parallelizes operators with window policies that require forward context (cf. Section 2.4). The splitter does not only determine the window extents on the input stream, but also when a window computation is triggered (i.e., when the `query` function is called.). The actual computation of the window results, i.e., the execution of the query function, is then scheduled to one of the available operator instances.

Approaches implementing Pane-based Splitting. Mencagli et al. [103, 104] split large panes into sub-panes with a proportional-integrative-derivative controller (PID) that automatically adjusts the splitting threshold. Their work focuses on burstiness in event arrival rates as to avoid bottlenecks.

Balkesen et al. [13] parallelize sliding window processing on data streams with pane-based splitting. A ring-based pane-partitioning splits the input stream and assigns panes to processing nodes that are logically ordered in a ring. The ring structure eases ordered processing and reduces communication overhead. Their approach first assigns windows to the ring-ordered nodes round-robin. Each node then processes those panes that belong to its assigned windows. A pane result calculated on one node can then be easily forwarded to the next node where the next window this pane belongs to assigned. A merge node aggregates the pane-results whereby it allows for a constrained degree of disorder to balance efficiency and output quality.

Koliousis et al. propose SABER [78], an SP engine that manages query processing on heterogeneous hardware with CPU and GPU cores. A splitter first splits the incoming event streams into batches of a fixed size and assigns them to a processing unit, i.e., a CPU core or a GPU processor. Each processing unit executes a "query task", i.e., a function that takes *n* batches, one from each of the *n* incoming event streams and calculates a function on those *n* batches in parallel applying an *n*-ary operator function. SABER aims to keep the size of the batches independent from the window policy, i.e., from the window size and slide. To this end, the operator function is divided into a fragment operator function and an assembly operator function. The fragment operator function defines the processing of a sequence of window fragments and produces window fragment results. The assembly operator function constructs the complete window result by combining the window fragment results. This way, different from classical pane-based splitting, the size of the processed batches is independent of the window slide.

3.2 Elasticity

This section discusses elasticity solutions for SP systems. It starts with centralized approaches where a single controller adapts the parallelism for the complete operator graph. Then, we discuss decentralized approaches where multiple controllers — each responsible for a sub-set of operators— adapt the parallelism. Table 2 gives an overview of the approaches according to the properties described in Section 2.5. A tick marks that an elasticity approach provides the property. These properties are: (1) Does the approach provide real-time guarantees? (2) Does it use a model rather than a threshold (only), (3) Is it a distributed approach? (4) Does it consider state migration?,(5) Does it evaluate its actions to improve its decision making process? Approaches without a tick either exclude these properties or do not consider them further.

3.2.1 Centralized Elasticity Solutions. This section presents elasticity solutions that implement a centralized controller. To further group the approaches, it begins with threshold-based approaches and continues with approaches that rely on a model for their scaling decision.

Threshold-based approaches. The following approaches reactively scale up or out when a threshold is met.

Satzger et al. [122] propose a distributed SP platform in Erlang called ESC for balanced, fault tolerant elastic stream processing in homogeneous cloud-environments. With this platform, programmers can adapt policies for thresholds, policies for scaling and splitter rules where the thresholds are set for workload and queue length. Their platform follows the vision of autonomic computing [74]. An "autonomic manager" controls the number of VMs and the distribution of operator instances on them. The contribution of ESC is rather its extensible architecture than the specific methods for elasticity control. While operators can be stateful, the authors do not describe a state migration process. They further leave interference effects of multiple operators on a shared node as future work.

Name, Ref	Data: System	Data:Workload	Timing: RE / PR	Objective	Realtime	Scale Up / Out	Model-based	Distributed	State Migration	Evaluation	Infrastructure	Year
Schneider [123]	x		RE	TP		U		x		x	SM	2009
Satzger [122]	x		RE	Lat		O					Cloud	2011
Gulisano [57]	x		RE	Res.		U			x		Cloud	2012
Tang [133]	x		RE	TP		U	x		x	x	SM	2012
Fernandez [44]	x	x	RE	TP		O			x		Cloud	2013
Balkesen [14]	x		PR	Lat	x	O	x				Cluster	2013
Akidau [3]	x	x	RE	LB, Lat		U, O			x		Cluster, Cloud	2013
Gedik [52]	x		RE	TP		O		x	x	x	Cluster	2014
Kumbhare [82]	x	x	PR	TP		O	x				Cloud	2014
Lohrmann [91]	x		RE	Lat, TP	x	O	x	x			Cluster	2014
Heinze [61]	x		RE	Util, Lat		O	x*		x	x	Cluster	2014
Heinze [59]	x		RE	Migr., Lat	x	O	x		x	x	Cluster	2014
Heinze [62]	x		RE	Lat	x	O	x		x	x	Cluster	2015
Lohrmann [90]	x	x	PR	Lat	x	O	x			x	Cluster	2015
Mayer [96]		x	PR	Lat	x	O	x	x			Cluster	2015
Zacheilas [152]		x	PR	Lat	x	O	x			x	Cloud	2015
Sun [131]	x		PR	Lat		O	x				Cloud	2015
Hochreiner [69]	x		RE	Lat		O		x			Cloud	2016
Hochreiner [68]	x	x	RE	Lat		O			x		Cloud	2016
Mencagli [102]		x	RE	TP		O	x	x			Cluster	2016
De Matteis [34, 37]	x	x	PR	Lat	x	U	x		x		SM	2016
De Matteis [35]	x	x	PR	Lat	x	O	x		x		Cluster	2017
Hidalgo [63]	x	x	R/P	TP		U	x			x	Cluster	2017
Kombi [79]	x	x	PR	TP, Stab.		U, O	x		x		Cloud	2017
Cardellini [23]	x	x	PR	Lat		U, O	x*			x	Cloud, Fog	2018
Mencagli [103]	x	x	PR	LB, Lat		U, O	x		x		Cloud	2018

Table 2. Categorization of SP operator elasticity literature. The first column contains the first author and the reference. The other columns specify the categorization according to the criteria introduced in Section 2.5. We ordered the table by publication date (last column). Abbreviations: *RE or PR* : reactive or proactive approach; Objectives (Obj.): *TP*: Throughput, *Lat*: Latency, *Res*: Resources, *LB*: Load Balancing, *Util*: Resource Utilization, *Migr*: Number of migrations, *Stab*: Number of re-configurations; *RT*: real-time guarantees; *Ev.*: evaluation of scaling effects; *SM*: Single Machine; *: Model-based with learning.

The elasticity component in VISP by Hochreiner et al. [68] uses thresholds on queue size or delay for each operator. VISP is a distributed SP system specialized on Internet of Things applications. It provides data integration, a “marketplace” for operators and operator topologies, and billing. Special is a shared key-value store for state that prevents state migrations.

With an upper and lower threshold for CPU utilization, in “StreamCloud" by Gulisano et al. [57] an “Elasticity Manager" reactively controls the average CPU utilization of the cluster hosting the operator graph. The authors propose the elasticity management in addition to a new key-based parallelization technique as described in Section 3.1. Their goal is to minimize resource consumption of an SP system in a private cloud to free resources for other applications. To scale out fast, a resource manager provides a pool of idle machines that can be activated when needed. To manage state, the authors propose an overlap phase and a state migration protocol. The involved operator instances agree on a start time where the former and the now responsible operator instance either start to process the same data until the state can be discarded at the old instance (overlap phase for sliding windows) or migrate state and buffer input data until the migration is finished (migration).

Similarly, Fernandez et al. [44] reactively scale out their SP system based on CPU thresholds: Their elasticity mechanism heuristically removes throughput bottlenecks where the average *user and system* CPU utilization exceed a threshold. The system CPU utilization tells about the real load on the machine, possibly induced by other, interfering applications. Again, the solution increases the scale out speed by keeping a "pool" of idle VMs. To easily migrate state and be fault tolerant, the implementation frequently stores checkpoints of operator states. Scale in, i.e. merging of states, is mentioned as future work. Madsen and Zhou present in [93] a similar approach to reduce the latency induced by the state migration with re-using availabe checkpoints.

The MillWheel framework proposed by Akidau et al. [3] scales up or out based on thresholds on CPU or memory utilization. The framework provides a fault tolerant, exactly once semantic for highly scalable SP applications and key-based, elastic data parallelization. Consistent state migration is supported with atomic state write operations and tokens that enforce single writer semantics.

Reactive Model-based approaches. With a greedy algorithm, Tang and Gedik [133] dynamically adapt the level of pipelining of an SP system on a multi-core machine. The authors use the assumptions that less utilized threads have a higher throughput and that pipelining and task parallelism are inherently available in the operator graph. Their solution minimizes the overall utilization for all threads. For fast processing, an exhaustive tree based aggregation reduces the search space of the algorithm. To continuously improve, the approach reverses and blacklists adaptations that do not sufficiently improve throughput. To migrate state, the system blocks while spawning new threads.

The group around Thomas Heinze published three approaches in the topic of elastic scaling. The first compares different auto-scaling strategies. The second tackles the problem of latency spikes due to state migration and the third tackles the question how to find optimal system parameters to configure an elastic SP system. Heinze et al. [61] analyze three different auto-scaling strategies on a cluster: global thresholds, local thresholds and reinforcement learning (RL). The goal is to maximize the utilization of processing nodes (i.e. to reduce the number of required nodes) while keeping the latency low. They use their system FUGU [60] for evaluations. The global threshold strategy scales based on to the average CPU utilization of the cluster, the local based on the CPU utilization per processing node. The rewards used in the RL approach depend on the degree the host utilization differs from a target utilization. The authors emphasize the importance to update the RL model according to experienced success or failure. The comparison shows that global threshold methodologies are not feasible for fast adaption of SP systems and that the RL-approach leads to the lowest latency with the highest utilization.

The latency spike a state migration induces is the focus of the follow up work by Heinze et al. [59]. They propose a reactive model-based controller that scales their SP system and guarantees an average end-to-end latency bound while keeping the resource utilization high. Their approach differentiates mandatory and optional migrations. Mandatory are migrations due to node overload (scale out). Optional migrations are due to node underload to improve the overall node utilization (scale in). These optional migrations are carried out only when the expected latency spike due to migration does not violate the latency bound. To predict the spike, the algorithm considers the size of the state to migrate, the total numbers of operators to move and the current arrival rate of the system. The latency values used for the cost model are calculated from the expected queue lengths if the system pauses. To improve the decision making, the system continuously updates its expected pausing times with measured values from performed migrations. In case of an optional migration, if releasing a complete host violates the latency bound, operators are migrated stepwise until the host can be shut down. The authors employ the FLUX-state migration protocol [127]. Their bin packing algorithm (cf. [60]) reassigns the selected operators to migrate to new machines based on CPU utilization.

Finally, Heinze et al. [62] handle the problem that elasticity strategies require user defined parameters, for example sampling frequencies, thresholds and placement strategies. As these parameters significantly influence the system's performance and cost, they need to be set carefully. The authors propose a model-based parameter-optimization method. With simulations, it automatically tunes six different thresholds to achieve cost-optimality with Heinze et al.'s reactive scaling-method.

Proactive Model-based approaches. Balkesen et al. [14] propose a multi-query SP framework with key- and pane-based data parallelization that manages how input streams are assigned to splitter nodes. Their goal is to minimize the number of processing nodes while keeping end-to-end latency bounds and balanced load. With an exponential smoothing technique and user-provided meta-data, they predict the future arrival rate and the change behavior of the input stream. Given this data, they calculate the number of required splitter nodes and assign them the input streams with a packing algorithm. Additionally, a mathematical equation is used to calculate the required parallelization degree given the predicted workload, node capacity and processing cost of input data items. A pool of idle nodes enables fast scale out. To add an operator instance, splitter and merger stop to connect to the new node. The approach does not include state migrations.

Sun et al. [131] propose Re-Stream, a scheduler for distributed and parallel SP systems that is both latency- and energy-aware. A re-scheduling algorithm minimizes the response time of operators on the critical path and schedules the other operators to minimize energy consumption of the system.

Solutions that employ the Kingman's formula from Queuing Theory (QT) [76] to calculate parallelization degrees come from Lohrmann et al. [90] and De Matteis and Mencagli [34, 37]. This formula provides average queuing delays for general arrival and processing distributions (G/G/x queues) under heavy load. Worst case analysis with QT (cf. Mayer et al. [96]) is possible for specific distributions only.

Lohrmann et al. [90] extend their system Nephele-streaming [91] we describe in Section 3.2.2 with a centralized controller to guarantee an average latency bound while minimizing resource consumption. With Kingman's formula, they predict queuing latencies and adapt the data parallelization degree of the operators in the SP system accordingly. To quickly react to sudden workload burst, a reactive component additionally doubles the parallelization degree if an operator becomes a bottleneck. The authors assume homogeneous processing nodes. Concerning state migration techniques they refer to other publications in this field.

De Matteis and Mencagli [34, 37] propose latency-aware and energy-efficient scaling of key-based data parallel SP operators. With a Model Predictive Control strategy, they control queuing latencies and energy consumptions on multi-core machines. Their goal is to minimize a latency-violation penalty, energy-consumption cost and consider a system stability value to avoid oscillation. Similar to the work of Mayer et al. [96], a disturbance forecaster predicts the future arrival rate and processing time. As in Lohrmann et al. [90], De Matteis and Mencagli predict the average per-tuple latency with QT. Moreover, De Matteis and Mencagli propose a power consumption model that predicts energy consumption for a CPU frequency and number of active cores. A state migration protocol exchanges state with a shared memory state storage. Special about this migration protocol is that only those instances pause their processing that are involved in the state migration. The other instances continue processing. Later, De Matteis and Mencagli extend their approach to horizontal scaling across several machines [35], building on the same predictive control model.

Zacheilas et al. [152] use a shortest path algorithm to proactively scale applications written with their SP framework (cf. [158]). Their algorithm minimizes resource cost, penalties for missed tuples due to operator overload and the state migration downtimes. The approach predicts latency and workload using a Gaussian Process and then models possible system configurations as a directed, acyclic graph. A shortest path algorithm finds the cost minimal sequence of configurations.

Hidalgo et al. [63] propose a hybrid reactive and proactive elasticity controller implemented on top of the S4 SP system [110]. The elasticity controller has two parts, a reactive short-term adaptation and a proactive mid-term adaptation. It controls the load of an operator based on its workload-to-throughput ratio and aims to maximize the system's throughput, avoiding bottlenecks. The short-term adaptation reactively changes the parallelization degree of an operator based on utilization-thresholds. The mid-term adaptation predicts the future load-state of an operator with a Markov-Chain model. It adapts the parallelization degree to the most probable load-state. The authors mention possible state migration solutions that can be included into their approach.

Kombi et al. [79] published the AUTOSCALE approach that centrally adapts the parallelization degrees of the operators in an operator graph based on predictive input values. Their goal is to proactively avoid congestion within the operator graph but be resource optimal and avoid too frequent reconfigurations. For each operator, AUTOSCALE predicts its future input in two ways: First, using linear regression, it predicts the input event rate of each operator using data from the last monitoring interval. Second, it predicts the input event rate from the predicted output event rate of the upstream operators and its selectivity. To decide about adapting the parallelization degree of a given operator, they combine these two input-estimations and calculate an activity metric from the input and the operators processing capacity. The scaling decisions then consider the activity metric and the derivative of the linear regression function where the letter serves as trend-indicator for the input load. While the solution considers stateful operators, the authors do not discuss state migration.

Cardellini et al. [23] propose a hierarchical controller for a distributed SP system to manage the parallelization degree and placement of operators. Local components send elasticity and migration requests to a global component that prioritizes and approves the requests based on benefit and urgency of the requested action. The cost-metric the global controller minimizes comprises the downtime caused by an action, the performance penalty in case of overloaded operators, and the cost for required resources. The global approval is made using a token bucket implementation. Regarding elasticity, the authors propose two concepts: A CPU-threshold based and a reinforcement learning based one. The reinforcement concept is further split up in a basic and a model supported solution. The basic solution switches between exploitation and exploration phases. The model supported solution pre-computes possible long-term costs based on the probabilities for parallelization degrees and arrival rates. It updates its knowledge at runtime which supersedes an exploitation phase. Their implementation in D-Storm (cf. [88]) migrates state with downtimes. Due to its decentralized nature, the controller is applicable in widely distributed environments.

Mencagli et al. [103] propose a two-level autonomic adaptation system for pane-based SP operators with two control loops: An inner loop schedules incoming events to worker threads to balance load at bursty data arrival rates. An outer loop controls the number of workers for long-term trends in the average data arrival rate. The authors argue that the mathematical models for conventional Control Theory methods are too complex given a system with multiple components and interdependent dynamics. Hence, they apply a Fuzzy Logic Controller which a domain expert configures. Several synthetic and real-world scenarios show the Controllers ability to deal with fluctuating workload. In case of changes in the thread level, the system updates the information that ensures that workers consistently assemble the pane results in the second stage of the pane-based operator.

Most of the elasticity controllers assume that the performance of allocated compute resources is stable. However, Kumbhare et al. [83] observe performance variations in VMs on multi-tenant clouds that require frequent adaptation of the configuration of the SP system (elasticity and placement of operator components). They propose a predictive controller to dynamically re-plan the allocation of elastic cloud resources which mitigates the impact of both resource and workload fluctuations.

3.2.2 Distributed Elasticity Solutions. There are a couple of papers that explore *distributed* elasticity controllers in charge of observing and controlling different sub-parts of the operator graph. We group them into approaches where a controller is responsible for multiple operators and approaches that focus on the control of single operators.

Multi-Operator Solutions. Some solutions find global consensus where the controllers communicate in order to find agreements in their reconfiguration choices. Early work by Weigold et al. [142] proposes a *rule-based* autonomic controller for distributed components in grid computing. The authors propose an architecture and discuss some implications of their design choices, but do not explicitly state how to configure the controller (i.e., the rules) in order to achieve specific optimization goals. Mencagli [102] proposes a distributed solution that employs a *game-theoretic* controller for each SP operator. By starting a game, each local controller determines the optimal parallelization degree for its operator, thereby maximizing the throughput while minimizing the monetary cost. Mencagli compares a non-cooperative and an incentive-based approach that encourages cooperation. The latter leads to a better solution for the whole system than the non-cooperative approach. Additionally, Mencagli et al. [101, 106] examine distributed elasticity control by applying *Model Predictive Control* in combination with a *cooperative optimization* framework. In particular, the effects of *switching costs* between configurations is modeled by a mathematical function which is used by a proactive control strategy to globally optimize the elasticity decisions. To find agreement between the distributed controllers, they apply the distributed subgradient method to optimize the sum of cost functions over all operators.

Lohrmann et al. [91] propose Nephele-Streaming, an SP framework that uses micro batching of data items. The authors reactively balance throughput and latency, aiming to keep throughput high but keep the average end-to-end latency withing user-defined bounds. They therefore use dynamic output buffer sizes ("adaptive output buffer sizing") and task fusion ("dynamic task chaining"). The former reduces the output batch size and thus waiting latencies. The latter increases the number of tasks that run within the same thread to reduce communication latencies ("reverse pipelining"). They propose a distributed, model-based approach with a set of QoS managers. Each manager manages the latency constraints for one part of the operator graph based on latency and CPU load measurements. When fusing tasks, i.e. omitting queues between them, the system either drops the data items in these queues or waits until the data items in the queues are processed. Their solution targets big cluster of nodes, which motivates the decentralized approach.

Single-Operator Solutions. This section summarizes three threshold and one model based approaches that provide elasticity control for single operators (or a limited sequence of operators) only instead of the complete operator graph. Schneider et al. [123] provide an algorithm that adapts the thread level of a single SP operator to control its throughput. The algorithm greedily adapts the thread level until throughput is not increased further. Then a stability condition is met and the thread level is kept. If the system detects changes in the throughput, it re-runs the algorithm to adapt the thread level according to changes in the workload as well as higher external load on the system. Noteworthy is that the authors provide one global queue the threads share as input queue. They further explicitly consider interferences on the node with other processes and adapt the thread level accordingly. They assume a single node environment.

Gedik et al. [52] propose an extension of their key-based data parallelization framework [124] that reactively adapts the parallelization degree of parallel regions according to changes in the workload. Their goal is high throughput at low resource usage in a best effort manner. A threshold based congestion-detection model uses back pressure information to decide about scaling. The splitter detects congestion and measures throughput of its operator. To avoid frequent scaling, the elasticity component remembers effective operating states of the system (parallelization degree) for the current workload. When the workload changes, the remembered system states are purged, and the system settles again to the new workload. A level function defines how many instances to add or remove. Setting the level function can make the adaptation more or less aggressive. To migrate state, instances store it in a database to make it available to newly spawned operator instances. During migration, the splitter stops. The proposed consistent hash function minimizes required state migration and ensures load balancing. To apply this solution in a multi-operator graph, one provides one splitter per parallel region (cf. [124], Section 3.1.3).

Hochreiner et al. [69] propose a reactive elasticity controller for their distributed SP platform PESP. For each operator, the controller minimizes *monetary* costs while ensuring moderate queuing times and CPU utilization with threshold driven scaling. The optimization algorithm considers real cloud provider pricing models that include a minimum time a VM needs to be rented. If an instance shall be shut down due to over provision, the controller selects the instance that has the least left time already paid for. With state being stored in a shared directory, operator scaling does not require state migration.

A proactive, model-based approach comes from Mayer et al. [96]. They provide a *window*-based data parallelization single operator framework. The operator instances can thereby be distributed to multiple machines. An elasticity controller proactively ensures a *worst case* latency limit using a queuing-theory based model. Thereby, all operator instances are assumed to behave homogeneously, i.e. have the same service times. While this model is limited to specific probabilistic distributions of arrival and processing rates, it enables the worst case limitation of queuing latency. In their framework, windows, when assigned to an operator instance once, will not be migrated. If an instance shall be shut down, it finishes the processing of its assigned windows and shuts down afterwards. Hence, scaling does not require state migration or downtime.

4 RELATED WORK

While this survey focusses on SP operator parallelization and elasticity, there are further important aspects when it comes to manage SP systems. This section provides a brief overview and discussion. The following aspects are discussed: Placement of SP operators, efficient resource provision, and efficient processing algorithms for SP operators. Furthermore, we discuss the relation of SP systems to batch processing systems.

4.1 Operator Placement

One important aspect in SP system management is the placement of SP operators in a distributed infrastructure. In the realm of SP system, this problem is often referred to as the *scheduling problem*, i.e., where to execute which parts of the operator graph. This regards the coarse-plained operator placement in different domains, e.g., in different data centers, but also the fine-grained placement within a single domain, e.g., within a single data center. Due to dynamic workloads of an SP system, operator placement cannot be a static decision, but operators need to be frequently migrated. In this section, we discuss some general challenges and solutions to this end.

Cardellini et al. [21] formulate the placement problem as an integer linear program (ILP) that can be solved optimally with an ILP solver. However, due to the NP-completeness of the problem, there is the need for efficient heuristics. SBON by Pietzuch et al. [113] is a placement algorithm that is based on a spring relaxation model and optimized the network utilization. Rizou et al. [119] propose another placement algorithm that uses the gradient descent method and optimizes the network utilization. SODA [143] by Wolf et al. is an early scheduler proposed for IBM's System S. To neither overload the network nor processing capacities of the computing nodes, SODA periodically admits or rejects new operator graphs and places them optimally. Amini et al. [5] propose a two-tiered scheduler that maximizes system throughput. The first tier decides about long-term operator placement, while the second tier reacts to bursts in workload with short-term CPU scheduling and event flow control on each computing node. When sources and sinks in the operator graph are mobile, the optimal operator placement changes over time, so that operators need to be migrated. To this end, MigCEP [112] by Ottenwälder et al. performs a plan-based migration of operators by predicting mobility patterns. Operator migration can in some cases be too expensive or infeasible. Xing et al. [148] have developed a placement algorithm that provides *resilient* placements that can withstand workload changes without operator migrations. Similarly, Drougas and Kalogeraki [40] propose a method for operator placement and parallelization that is resilient to sudden *bursts* in the input streams.

With the increasing need for operator parallelization and elasticity, fine-grained placement, i.e., where to place which operator components, has become an urgent problem. Aniello et al. [7] propose an adaptive online scheduler tailored to the Storm ESP system that takes into account traffic patterns between components of the ESP system to reduce inter-node traffic. The T-Storm scheduler by Xu et al. [149] follows a similar goal; in comparison to Aniello et al. [7], T-Storm introduces a couple of further optimizations tailored to the Storm SP system. Fischer and Bernstein [46] model the communication between the instances in Storm as a graph and solve a graph partitioning problem to minimize the communication while keeping the computational load between the nodes balanced. Their approach shows a better performance than the aforementioned adaptive online scheduler by Aniello et al. [7].

There are also combined approaches which regard determining the operator parallelization degree and the fine-grained placement of operator instances as a holistic problem. Cardellini et al. [22] formalized the combined problem for data-parallel SP as an integer linear program (ILP) that can be solved optimally with an ILP solver. P-Deployer [89] by Liu and Buyya models the same problem as a bin-packing problem and solves it by a heuristic that is based on the first fit decreasing method. Backman et al. [11] propose a scheduling framework that can exploit data and task parallelism in SP operators. It minimizes the end-to-end latency with operator parallelism and the scheduling of the corresponding operator components on the available computing node. The authors use a tiered bin-packing problem, which allows for prioritizing operators, and solve the optimization problem using simulation-based latency estimation. Madsen et al. [94] provide a solution to integrate load balancing, collocating (i.e., placement), and horizontal scaling (i.e., determining the parallelization degree of operators). They model the problem as a Mixed-Integer Linear Program, solved with a heuristic greedy algorithm. While the papers discussed in this paragraph have some overlap with the work on operator elasticity discussed in Section 3.2 their focus is more on the placement problem than on elasticity control.

4.2 Efficient Resource Provisioning

Another aspect that is related both to parallelization and elasticity is how to efficiently provide the required resources in cloud environments.

The SP system Stela by Xu et al. [150] tackles the problem which operator to scale out when there are more resources added to the system, and which operators to scale in when resources are removed. Stela maximizes the post-scaling throughput by scaling out the operators that have the highest predicted impact on the application throughput based on an analysis of the congestion in the operator graph.

Borkowski et al. [17] tackle the problem of minimizing the number of superfluous scaling activities at highly noisy workloads. To this end, they employ non-linear filtering techniques from the field of signal processing.

Lombardi et al. [92] make the observation that the scaling of operators and the scaling of resources are two independent tasks that do not necessarily have to be performed jointly. They propose the ELYSIUM controller that first adapts the parallelization degree for each operator, and then adapts the resource provisioning only when needed. To this end, they leverage a resource estimator that predicts the resource consumption based on the current resource utilization. Similarly, Van der Veen et al. [138] propose a controller to automatically adjust the number of virtual machines assigned to a deployment of the Storm ESP system.

4.3 Efficient Operator Execution and Re-Use

Besides questions of scaling and placement of SP operators, the efficient execution of operators and whole operator graphs has received a lot of attention in the literature. On the one-hand side, the execution of single operators is optimized by efficient processing algorithms. On the other-hand side, the execution of overall operator graphs is optimized by operator re-use among multiple concurrent queries.

ZStream by Mei et al. [100] implements a tree-based pattern-plan structure and dynamically finds the optimal plan to evaluate the CEP-pattern.

The work of Poppe et al. [114] manages the trade-off between memory consumption and processing throughput when detecting sequences of arbitrary, statically unknown length, e.g. Kleene-closure queries. The approach stores common subsequences of multiple pattern instances for re-use, to decreasing the processing load at the cost of memory consumption.

There is a large body of work on the efficient incremental processing of general queries on sliding windows in SP operators. The tutorial by Hirzel et al. [66] provides a comprehensive overview. Tangwongsan et al. in [135] and [134] optimize processing depending on the mathematical properties of the operator function (invertability, associativity and commutativity). Le-Phuoc et al. [85] propose an incrementally sliding window approach for the parallel processing of multiway join and aggregation operations.

When there are multiple queries in an SP system, re-use of results from one query in another query is a common method to reduce overhead. The RECEP system by Ottenwälder et al. [111] tackles multi-query SP systems in mobile scenarios, where there are many similar queries that show overlapping ranges of interest in time and space. RECEP allows to re-use results of similar queries, guaranteeing a user-defined quality requirement with precision and recal. SQPR by Kalyvianaki et al. [71] and SPASS by Ray et al. [116] are placement optimizers that leverage the sharing of computations between the sub-patterns of multiple queries. SlickDeque by Shein et al. [128] improves throughput and latency of incremental invertible and noninvertible SP operators and supports efficient multi-query processing.

Verner et al. [139] propose a deadline-aware scheduler for hybrid compute platforms for operators that process multiple different input streams in a highly-parallel fashion. Their solution consist of multiple CPUs and a single accelerator (such as a GPU).

4.4 Batch Processing Systems

Modern batch processing systems make heavy use of data parallelism. Those systems are often based on the MapReduce paradigm presented by Dean and Ghemawat [38], that splits the input data with keys. A couple of works push the traditional MapReduce more toward a streaming behavior. In [84], Lam et al. present Muppet, a MapReduce modification where an “update” function" replaces the reduce function. The update function makes intermediate results accessible any time, giving the impression of a results stream. Similarly, Hadoop Online by Condie et al. [27, 28] allows users to see “early returns”, i.e., intermediate results of the reduce function. Stream MapReduce by Brito et al. [19] introduces “windowed reducers” to output a stream of results according to a window policy. Kumbhare et al. [81] extend Stream MapReduce by methods for adaptive load-balancing, runtime elasticity and fault tolerance. Beyond approaches to adapt a streaming model in MapReduce, Apache Flink [20], Apache Spark [154] and AJIRA [137] support batch and stream processing.

5 CONCLUSION AND OUTLOOK

As the rate of publications on parallelization and elasticity in SP is still increasing, we see the demand for a structured overview of this field. In this survey, we discussed and classified solutions for parallelization and elasticity in SP systems to provide a comprehensive overview. Besides providing an overview, with our work we hope to enhance the mutual understanding of research communities that look at SP systems from different angles. For instance, we noticed that researchers from the general SP domain typically assume SP operators that have key-partitioned state. In contrast, researchers from the CEP domain focus on window-based operations. Hence, the solutions for parallelization and elasticity that are proposed from the different domains are different. For future research in parallel and elastic SP, we observe a couple of recent trends that will impact SP systems. These general trends are fog and edge computing, in-network computing (e.g., on smart NICs), and sophisticated cloud cost models.

As we found in this survey, most of the current solutions on SP parallelization and elasticity are developed for homogeneous resources, often provided by a cloud data center. As a future work, we encourage an extension towards heterogeneous resources, especially considering the upcoming trend toward fog and edge computing that comes with more heterogeneous processing nodes [16]. Further, fog and edge compute nodes are limited in their computational capabilities, so that the “illusion of unlimited hardware” provided by cloud computing may not hold at the edge. Here, load shedding or approximate computing may be unavoidable, and we see first methods for approximate SP being proposed [115].

We see an emerging field of new programmable network devices that allow for offloading computing from CPUs to the network, along with network programming languages such as P4. Early work on in-network CEP points to tremendous potential of in-network computing for this domain, but also reveals challenges due to limitations both in the programmable hardware as well as in the network programming language [77].

Finally, promising QoS-metrics to consider in elasticity approaches are energy consumption to support environmental friendly IT solutions and the cost models of cloud providers that go beyond the standard "pay-as-you-go" model. These models support, e.g, "spot instances" and thus make room for financial savings [156].

REFERENCES

[1] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (Aug. 2003), 120–139. DOI: https://doi.org/10.1007/s00778-003-0095-z

[2] Asaf Adi and Opher Etzion. 2004. Amit - the Situation Manager. *The VLDB Journal* 13, 2 (May 2004), 177–203. DOI: https://doi.org/10.1007/s00778-003-0108-y

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. 2013. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.* 6, 11 (Aug. 2013), 1033–1044. DOI: https://doi.org/10.14778/2536222.2536229

[4] Elias Alevizos, Alexander Artikis, and George Paliouras. 2017. Event Forecasting with Pattern Markov Chains. In *Proc. of the 11th ACM Int. Conf. on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 146–157. DOI: https://doi.org/10.1145/3093742.3093920

[5] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. 2006. Adaptive Control of Extreme-scale Stream Processing Systems. In *26th IEEE Int. Conf. on Distributed Computing Systems (ICDCS'06)*. 71–71. DOI: https://doi.org/10.1109/ICDCS.2006.13

[6] H. Andrade, B. Gedik, K.-L. Wu, and P.S. Yu. 2011. Processing high data rate streams in System S. *J. Parallel and Distrib. Comput.* 71, 2 (Feb. 2011), 145–156. DOI: https://doi.org/10.1016/j.jpdc.2010.08.007

[7] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. 2013. Adaptive Online Scheduling in Storm. In *Proc. of the 7th ACM Int. Conf. on Distributed Event-based Systems (DEBS '13)*. ACM, New York, NY, USA, 207–218. DOI: https://doi.org/10.1145/2488222.2488267

[8] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142. DOI: https://doi.org/10.1007/s00778-004-0147-z

[9] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A View of Cloud Computing. *Commun. ACM* 53, 4 (April 2010), 50–58. DOI: https://doi.org/10.1145/1721654.1721672

[10] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. 1999. Balanced Allocations. *SIAM J. Comput.* 29, 1 (Sept. 1999), 180–200. DOI: https://doi.org/10.1137/S0097539795288490

[11] Nathan Backman, Rodrigo Fonseca, and Uğur Çetintemel. 2012. Managing Parallelism for Stream Processing in the Cloud. In *Proc. of the 1st Int. Workshop on Hot Topics in Cloud Data Processing (HotCDP '12)*. ACM, New York, NY, USA, 1:1–1:5. DOI: https://doi.org/10.1145/2169090.2169091

[12] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. 2013. Rip: Run-based intra-query parallelism for scalable complex event processing. In *Proceedings of the 7th ACM Int. Conf. on Distributed event-based systems*. ACM, 3–14. http://dl.acm.org/citation.cfm?id=2488257

[13] Cagri Balkesen and Nesime Tatbul. 2011. Scalable data partitioning techniques for parallel sliding window processing over data streams. In *Int. Workshop on Data Management for Sensor Networks (DMSN)*. http://www.softnet.tuc.gr/dmsn11/papers/paper03.pdf

[14] Cagri Balkesen, Nesime Tatbul, and M Tamer Özsu. 2013. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM Int. Conf. on Distributed event-based systems*. ACM, 15–26.

[15] P. Basanta-Val, N. Fernández-García, L. Sánchez-Fernández, and J. Arias-Fisteus. 2017. Patterns for Distributed Real-Time Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 28, 11 (Nov 2017), 3243–3257. DOI: https://doi.org/10.1109/TPDS.2017.2716929

[16] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. 2012. Fog Computing and Its Role in the Internet of Things. In *Proc. of the First Edition of the MCC Workshop on Mobile Cloud Computing (MCC '12)*. ACM, New York, NY, USA, 13–16. DOI: https://doi.org/10.1145/2342509.2342513

[17] Michael Borkowski, Christoph Hochreiner, and Stefan Schulte. 2018. Moderated Resource Elasticity for Stream Processing Applications. In *Euro-Par 2017: Parallel Processing Workshops. Lecture Notes in Computer Science*, Dora B. Heras et al. (Ed.), Vol. 10569. Springer, Cham, 5–16.

[18] Lars Brenna, Johannes Gehrke, Mingsheng Hong, and Dag Johansen. 2009. Distributed Event Stream Processing with Non-deterministic Finite Automata. In *Proc. of the Third ACM Int. Conf. on Distributed Event-Based Systems (DEBS '09)*. ACM, New York, NY, USA, 3:1–3:12. DOI: https://doi.org/10.1145/1619258.1619263

[19] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer. 2011. Scalable and Low-Latency Data Processing with Stream MapReduce. In *2011 IEEE Third Int. Conf. on Cloud Computing Technology and Science*. 48–58. DOI: https://doi.org/10.1109/CloudCom.2011.17

[20] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[21] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2016. Optimal Operator Placement for Distributed Stream Processing Applications. In *Proc. of the 10th ACM Int. Conf. on Distributed and Event-based Systems (DEBS '16)*. ACM, New York, NY, USA, 69–80. DOI: https://doi.org/10.1145/2933267.2933312

[22] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. 2017. Optimal Operator Replication and Placement for Distributed Stream Processing Systems. *SIGMETRICS Perform. Eval. Rev.* 44, 4 (May 2017), 11–22. DOI:

https://doi.org/10.1145/3092819.3092823

[23] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. 2018. Decentralized self-adaptation for elastic Data Stream Processing. *Future Generation Computer Systems* 87 (Oct. 2018), 171 – 185. DOI: https://doi.org/10.1016/j.future.2018.05.025

[24] Sharma Chakravarthy and Deepak Mishra. 1994. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering* 14, 1 (1994), 1–26. http://www.sciencedirect.com/science/article/pii/0169023X9490006X

[25] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. 2003. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*. ACM, New York, NY, USA, 668–668. DOI: https://doi.org/10.1145/872757.872857

[26] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B. Zdonik. 2003. Scalable Distributed Stream Processing.. In *CIDR*, Vol. 3. 257–268. http://nms.csail.mit.edu/papers/CIDR_CRC.pdf

[27] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. 2010. MapReduce online.. In *NSDI*, Vol. 10. 20. http://static.usenix.org/events/nsdi10/tech/full_papers/condie.pdf

[28] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. 2010. Online Aggregation and Continuous Query Support in MapReduce. In *Proc. of the 2010 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 1115–1118. DOI: https://doi.org/10.1145/1807167.1807295

[29] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language. In *Proceedings of the Fourth ACM Int. Conf. on Distributed Event-Based Systems*. ACM, 50–61. http://dl.acm.org/citation.cfm?id=1827427

[30] Gianpaolo Cugola and Alessandro Margara. 2012. Low latency complex event processing on parallel hardware. *J. Parallel and Distrib. Comput.* 72, 2 (Feb. 2012), 205–218. DOI: https://doi.org/10.1016/j.jpdc.2011.11.002

[31] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *Comput. Surveys* 44, 3 (June 2012), 1–62. DOI: https://doi.org/10.1145/2187671.2187677

[32] Marco Danelutto, Peter Kilpatrick, Gabriele Mencagli, and Massimo Torquati. 0. State access patterns in stream parallel computations. *The Int. Journal of High Performance Computing Applications* 0, 0 (0), 1–12. DOI: https://doi.org/10.1177/1094342017694134 arXiv:https://doi.org/10.1177/1094342017694134

[33] Marcos Dias de Assuncao, Alexandre da Silva Veith, and Rajkumar Buyya. 2017. Resource Elasticity for Distributed Data Stream Processing: A Survey and Future Directions. *arXiv preprint arXiv:1709.01363* (2017).

[34] 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 (PPoPP '16)*. ACM, New York, NY, USA, Article 13, 12 pages. DOI: https://doi.org/10.1145/2851141.2851148

[35] Tiziano De Matteis and Gabriele Mencagli. 2017. Elastic Scaling for Distributed Latency-Sensitive Data Stream Operators. In *2017 25th Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing (PDP)*. 61–68. DOI: https://doi.org/10.1109/PDP.2017.31

[36] Tiziano De Matteis and Gabriele Mencagli. 2017. Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach. *Int. Journal of Parallel Programming* 45, 2 (April 2017), 382–401. DOI: https://doi.org/10.1007/s10766-016-0413-x

[37] Tiziano De Matteis and Gabriele Mencagli. 2017. Proactive elasticity and energy awareness in data stream processing. *Journal of Systems and Software* 127 (2017), 302 – 319. DOI: https://doi.org/10.1016/j.jss.2016.08.037

[38] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of the 6th Conf. on Symposium on Operating Systems Design & Implementation - Volume 6 (OSDI'04)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1251254.1251264

[39] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, Walker M. White, and others. 2007. Cayuga: A General Purpose Event Monitoring System.. In *CIDR*, Vol. 7. 412–422. http://www.ccis.northeastern.edu/home/mirek/papers/2007-CIDR-CayugaImp.pdf

[40] Y. Drougas and V. Kalogeraki. 2009. Accommodating bursts in distributed stream processing systems. In *2009 IEEE Int. Symposium on Parallel Distributed Processing*. 1–11. DOI: https://doi.org/10.1109/IPDPS.2009.5161015

[41] Esper 2019. Esper. (jan 2019). Retrieved 2019-01-15 from http://www.espertech.com/

[42] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114–131. DOI: https://doi.org/10.1145/857076.857078

[43] R. C. Fernandez, P. Garefalakis, and P. Pietzuch. 2016. Java2SDG: Stateful big data processing for the masses. In *2016 IEEE 32nd Int. Conf. on Data Engineering (ICDE)*. 1390–1393. DOI: https://doi.org/10.1109/ICDE.2016.7498352

[44] 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 2013 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 725–736. DOI: https://doi.org/10.1145/2463676.2465282

[45] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2014. Making State Explicit for Imperative Big Data Processing. In *Proceedings of the 2014 USENIX Conf. on USENIX Annual Technical Conf. (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 49–60. http://dl.acm.org/citation.cfm?id=2643634.2643640

[46] L. Fischer and A. Bernstein. 2015. Workload scheduling in distributed stream processors using graph partitioning. In *2015 IEEE Int. Conf. on Big Data (Big Data)*. 124–133. DOI: https://doi.org/10.1109/BigData.2015.7363749

[47] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Kamp, and Michael Mock. 2017. Issues in complex event processing: Status and prospects in the Big Data era. *Journal of Systems and Software* 127, Supplement C (2017), 217 – 236. DOI: https://doi.org/10.1016/j.jss.2016.06.011

[48] Apache Software Foundation. 2015. Apache Storm Project Website. (2015). Retrieved Last Accessed 2019-01-15 from http://storm.apache.org

[49] The Apache Software Foundation. 2019. Apache Flink Project Website. (jan 2019). Retrieved 2019-01-15 from http://flink.apache.org/

[50] Buğra Gedik. 2014. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience* 44, 9 (2014), 1105–1128. DOI: https://doi.org/10.1002/spe.2194

[51] Buğra Gedik. 2014. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal* 23, 4 (Aug. 2014), 517–539. DOI: https://doi.org/10.1007/s00778-013-0335-9

[52] B. Gedik, S. Schneider, M. Hirzel, and K. L. Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1447–1463. DOI: https://doi.org/10.1109/TPDS.2013.295

[53] B. Gedik, H.G. Özsema, and Ö. Öztürk. 2016. Pipelined fission for stream programs with dynamic selectivity and partitioned state. *J. Parallel and Distrib. Comput.* 96 (2016), 106 – 120. DOI: https://doi.org/10.1016/j.jpdc.2016.05.003

[54] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*. ACM, New York, NY, USA, 151–162. DOI: https://doi.org/10.1145/1168857.1168877

[55] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. 2002. A Stream Compiler for Communication-exposed Architectures. In *Proceedings of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*. ACM, New York, NY, USA, 291–303. DOI: https://doi.org/10.1145/605397.605428

[56] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. 2016. Frames: Data-driven Windows. In *Proc. of the 10th ACM Int. Conf. on Distributed and Event-based Systems (DEBS '16)*. ACM, New York, NY, USA, 13–24. DOI: https://doi.org/10.1145/2933267.2933304

[57] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. 2012. StreamCloud: An Elastic and Scalable Data Streaming System. *IEEE Transactions on Parallel and Distributed Systems* 23, 12 (Dec. 2012), 2351–2365. DOI: https://doi.org/10.1109/TPDS.2012.24

[58] Thomas Heinze, Leonardo Aniello, Leonardo Querzoni, and Zbigniew Jerzak. 2014. Cloud-based Data Stream Processing. In *Proc. of the 8th ACM Int. Conf. on Distributed Event-Based Systems (DEBS '14)*. ACM, New York, NY, USA, 238–245. DOI: https://doi.org/10.1145/2611286.2611309

[59] 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 Int. Conf. on Distributed Event-Based Systems*. ACM, 13–22.

[60] Thomas Heinze, Yuanzhen Ji, Yinying Pan, Franz Josef Grueneberger, Zbigniew Jerzak, and Christof Fetzer. 2013. Elastic Complex Event Processing under Varying Query Load. In *First Int. Workshop on Big Dynamic Distributed Data (BD3)*. 25.

[61] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. 2014. Auto-scaling Techniques for Elastic Data Stream Processing. In *2014 IEEE 30th Int. Conf. on Data Engineering Workshops (ICDEW)*. IEEE, 296–302.

[62] 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 Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 276–287. DOI: https://doi.org/10.1145/2806777.2806847

[63] Nicolas Hidalgo, Daniel Wladdimiro, and Erika Rosas. 2017. Self-adaptive processing graph with operator fission for elastic stream processing. *Journal of Systems and Software* 127 (2017), 205 – 216. DOI: https://doi.org/10.1016/j.jss.2016.06.010

[64] Martin Hirzel. 2012. Partition and Compose: Parallel Complex Event Processing. In *Proc. of the 6th ACM Int. Conf. on Distributed Event-Based Systems (DEBS '12)*. ACM, New York, NY, USA, 191–200. DOI: https://doi.org/10.1145/2335484.

2335506

[65] Martin Hirzel, Scott Schneider, and Bugra Gedik. 2014. *SPL: An extensible language for distributed stream processing*. Technical Report. Research Report RC25486, IBM. http://hirzels.com/martin/papers/tr14-rc25486-spl.pdf

[66] Martin Hirzel, Scott Schneider, and Kanat Tangwongsan. 2017. Sliding-Window Aggregation Algorithms: Tutorial. In *Proc. of the 11th ACM Int. Conf. on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 11–14. DOI: https://doi.org/10.1145/3093742.3095107

[67] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A Catalog of Stream Processing Optimizations. *ACM Comput. Surv.* 46, 4 (March 2014), 46:1–46:34. DOI: https://doi.org/10.1145/2528412

[68] C. Hochreiner, M. Vogler, P. Waibel, and S. Dustdar. 2016. VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things. In *2016 IEEE 20th Int. Enterprise Distributed Object Computing Conf.* 1–11. DOI: https://doi.org/10.1109/EDOC.2016.7579390

[69] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar. 2016. Elastic Stream Processing for the Internet of Things. In *2016 IEEE 9th Int. Conf. on Cloud Computing (CLOUD)*. 100–107. DOI: https://doi.org/10.1109/CLOUD.2016.0023

[70] Waldemar Hummer, Benjamin Satzger, and Schahram Dustdar. 2013. Elastic stream processing in the Cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 3, 5 (Sept. 2013), 333–345. DOI: https://doi.org/10.1002/widm.1100

[71] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch. 2011. SQPR: Stream query planning with reuse. In *2011 IEEE 27th Int. Conf. on Data Engineering*. 840–851. DOI: https://doi.org/10.1109/ICDE.2011.5767851

[72] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. ACM, New York, NY, USA, 654–663. DOI: https://doi.org/10.1145/258533.258660

[73] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. 2017. A Holistic View of Stream Partitioning Costs. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1286–1297. DOI: https://doi.org/10.14778/3137628.3137639

[74] J. O. Kephart and D. M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan 2003), 41–50. DOI: https://doi.org/10.1109/MC.2003.1160055

[75] 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 Int. Conf. on Middleware (Middleware '09)*. Springer-Verlag New York, Inc., New York, NY, USA, Article 16, 20 pages. http://dl.acm.org/citation.cfm?id=1656980.1657002

[76] JFC Kingman. 1961. The single server queue in heavy traffic. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 57. Cambridge University Press, 902–904.

[77] Thomas Kohler, Ruben Mayer, Frank Dürr, Marius Maaß, Sukanya Bhowmik, and Kurt Rothermel. 2018. P4CEP: Towards In-Network Complex Event Processing. In *Proc. of the 2018 Morning Workshop on In-Network Computing (NetCompute '18)*. ACM, New York, NY, USA, 33–38. DOI: https://doi.org/10.1145/3229591.3229593

[78] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures. In *Proc. of the 2016 Int. Conf. on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 555–569. DOI: https://doi.org/10.1145/2882903.2882906

[79] Roland Kotto Kombi, Nicolas Lumineau, and Philippe Lamarre. 2017. A Preventive Auto-Parallelization Approach for Elastic Stream Processing. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th Int. Conf. on.* IEEE, 1532–1542. DOI: https://doi.org/10.1109/ICDCS.2017.253

[80] Sanjeev Kulkarni, Nikunj Bhagat, Maosong 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 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 239–250. DOI: https://doi.org/10.1145/2723372.2742788

[81] A. Kumbhare and others. 2015. Fault-Tolerant and Elastic Streaming MapReduce with Decentralized Coordination. In *2015 IEEE 35th Int. Conf. on Distributed Computing Systems*. 328–338. DOI: https://doi.org/10.1109/ICDCS.2015.41

[82] A. G. Kumbhare, Y. Simmhan, and V. K. Prasanna. 2014. PLAStiCC: Predictive Look-Ahead Scheduling for Continuous Dataflows on Clouds. In *2014 14th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*. 344–353. DOI: https://doi.org/10.1109/CCGrid.2014.60

[83] A. G. Kumbhare, Y. Simmhan, and V. K. Prasanna. 2014. PLAStiCC: Predictive Look-Ahead Scheduling for Continuous Dataflows on Clouds. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)(CCGRID)*, Vol. 00. 344–353. DOI: https://doi.org/10.1109/CCGrid.2014.60

[84] Wang Lam, Lu Liu, Sts Prasad, Anand Rajaraman, Zoheb Vacheri, and AnHai Doan. 2012. Muppet: MapReduce-style Processing of Fast Data. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1814–1825. DOI: https://doi.org/10.14778/2367502.2367520

[85] Danh Le-Phuoc, Hoan Nguyen Mau Quoc, Chan Le Van, and Manfred Hauswirth. 2013. Elastic and Scalable Processing of Linked Stream Data in the Cloud. In *Proc. of the 12th Int. Semantic Web Conf. - Part I (ISWC '13)*. Springer-Verlag New York, Inc., New York, NY, USA, 280–297. DOI:https://doi.org/10.1007/978-3-642-41335-3_18

[86] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Record* 34, 1 (2005), 39–44. http://dl.acm.org/citation.cfm?id=1058158

[87] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In *Proc. of the 2005 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '05)*. ACM, New York, NY, USA, 311–322. DOI:https://doi.org/10.1145/1066157.1066193

[88] Xunyun Liu and Rajkumar Buyya. 2017. D-Storm: Dynamic Resource-Efficient Scheduling of Stream Processing Applications. In *Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd Int. Conf. on.* IEEE, 485–492.

[89] Xunyun Liu and Rajkumar Buyya. 2017. Performance-Oriented Deployment of Streaming Applications on Cloud. *IEEE Transactions on Big Data* PP, 99 (2017), 1–1. DOI:https://doi.org/10.1109/TBDATA.2017.2720622

[90] B. Lohrmann, P. Janacik, and O. Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 399–410. DOI:https://doi.org/10.1109/ICDCS.2015.48

[91] Björn Lohrmann, Daniel Warneke, and Odej Kao. 2014. Nephele streaming: stream processing under QoS constraints at scale. *Cluster Computing* 17, 1 (March 2014), 61–78. DOI:https://doi.org/10.1007/s10586-013-0281-8

[92] Federico Lombardi, Leonardo Aniello, Silvia Bonomi, and Leonardo Querzoni. 2017. Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (2017), 1–1. DOI:https://doi.org/10.1109/TPDS.2017.2762683

[93] Kasper Grud Skat Madsen and Yongluan Zhou. 2015. Dynamic Resource Management In a Massively Parallel Stream Processing Engine. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15)*. ACM, New York, NY, USA, 13–22. DOI:https://doi.org/10.1145/2806416.2806449

[94] K. G. S. Madsen, Y. Zhou, and J. Cao. 2017. Integrative Dynamic Reconfiguration in a Parallel Stream Processing Engine. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 227–230. DOI:https://doi.org/10.1109/ICDE.2017.81

[95] C. Mayer, M. A. Tariq, R. Mayer, and K. Rothermel. 2018. GrapH: Traffic-Aware Graph Processing. *IEEE Transactions on Parallel and Distributed Systems* PP, 99 (2018), 1–1. DOI:https://doi.org/10.1109/TPDS.2018.2794989

[96] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2015. Predictable Low-Latency Event Detection With Parallel Complex Event Processing. *IEEE Internet of Things Journal* 2, 4 (Aug. 2015), 274–286. DOI:https://doi.org/10.1109/JIOT.2015.2397316

[97] Ruben Mayer, Christian Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2016. GraphCEP: Real-time Data Analytics Using Parallel Complex Event and Graph Processing. In *Proceedings of the 10th ACM Int. Conf. on Distributed and Event-based Systems (DEBS '16)*. ACM, New York, NY, USA, 309–316. DOI:https://doi.org/10.1145/2933267.2933509

[98] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. 2017. SPECTRE: Supporting Consumption Policies in Window-based Parallel Complex Event Processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conf. (Middleware '17)*. ACM, New York, NY, USA, 161–173. DOI:https://doi.org/10.1145/3135974.3135983

[99] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2017. Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing. In *Proceedings of the 11th ACM Int. Conf. on Distributed and Event-based Systems (DEBS '17)*. ACM, New York, NY, USA, 54–65. DOI:https://doi.org/10.1145/3093742.3093914

[100] Yuan Mei and Samuel Madden. 2009. ZStream: A Cost-based Query Processor for Adaptively Detecting Composite Events. In *Proc. of the 2009 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 193–206. DOI:https://doi.org/10.1145/1559845.1559867

[101] Gabriele Mencagli. 2016. Adaptive model predictive control of autonomic distributed parallel computations with variable horizons and switching costs. *Concurrency and Computation: Practice and Experience* 28, 7 (2016), 2187–2212.

[102] Gabriele Mencagli. 2016. A Game-Theoretic Approach for Elastic Distributed Data Stream Processing. *ACM Trans. Auton. Adapt. Syst.* 11, 2, Article 13 (June 2016), 34 pages. DOI:https://doi.org/10.1145/2903146

[103] Gabriele Mencagli, Massimo Torquati, and Marco Danelutto. 2018. Elastic-PPQ: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Generation Computer Systems* 79 (2018), 862 – 877. DOI:https://doi.org/10.1016/j.future.2017.09.004

[104] G. Mencagli, M. Torquati, M. Danelutto, and T. De Matteis. 2017. Parallel Continuous Preference Queries over Out-of-Order and Bursty Data Streams. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (Sept 2017), 2608–2624. DOI:https://doi.org/10.1109/TPDS.2017.2679197

[105] Gabriele Mencagli, Massimo Torquati, Fabio Lucattini, Salvatore Cuomo, and Marco Aldinucci. 2018. Harnessing sliding-window execution semantics for parallel stream processing. *J. Parallel and Distrib. Comput.* 116 (2018), 74 – 88.

DOI: https://doi.org/10.1016/j.jpdc.2017.10.021 Towards the Internet of Data: Applications, Opportunities and Future Challenges.

[106] Gabriele Mencagli, Marco Vanneschi, and Emanuele Vespa. 2014. A Cooperative Predictive Control Approach to Improve the Reconfiguration Stability of Adaptive Distributed Parallel Applications. *ACM Trans. Auton. Adapt. Syst.* 9, 1, Article 2 (March 2014), 27 pages. DOI: https://doi.org/10.1145/2567929

[107] Y. Nakamura, H. Suwa, Y. Arakawa, H. Yamaguchi, and K. Yasumoto. 2016. Design and Implementation of Middleware for IoT Devices toward Real-Time Flow Processing. In *2016 IEEE 36th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW).* 162–167. DOI: https://doi.org/10.1109/ICDCSW.2016.37

[108] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini. 2015. The power of both choices: Practical load balancing for distributed stream processing engines. In *2015 IEEE 31st Int. Conf. on Data Engineering.* 137–148. DOI: https://doi.org/10.1109/ICDE.2015.7113279

[109] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. 2016. When two choices are not enough: Balancing at scale in Distributed Stream Processing. In *2016 IEEE 32nd Int. Conf. on Data Engineering (ICDE).* 589–600. DOI: https://doi.org/10.1109/ICDE.2016.7498273

[110] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. 2010. S4: Distributed Stream Computing Platform. In *2010 IEEE Int. Conf. on Data Mining Workshops.* 170–177. DOI: https://doi.org/10.1109/ICDMW.2010.172

[111] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, Kirak Hong, and Umakishore Ramachandran. 2014. RECEP: Selection-based Reuse for Distributed Complex Event Processing. In *Proc. of the 8th ACM Int. Conf. on Distributed Event-Based Systems (DEBS '14).* ACM, New York, NY, USA, 59–70. DOI: https://doi.org/10.1145/2611286.2611297

[112] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. 2013. MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing. In *Proc. of the 7th ACM Int. Conf. on Distributed Event-based Systems (DEBS '13).* ACM, New York, NY, USA, 183–194. DOI: https://doi.org/10.1145/2488222.2488265

[113] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. 2006. Network-Aware Operator Placement for Stream-Processing Systems. In *22nd Int. Conf. on Data Engineering (ICDE'06).* 49–49. DOI: https://doi.org/10.1109/ICDE.2006.105

[114] Olga Poppe, Chuan Lei, Salah Ahmed, and Elke A. Rundensteiner. 2017. Complete Event Trend Detection in High-Rate Event Streams. In *Proc. of the 2017 ACM Int. Conf. on Management of Data (SIGMOD '17).* ACM, New York, NY, USA, 109–124. DOI: https://doi.org/10.1145/3035918.3035947

[115] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. StreamApprox: Approximate Computing for Stream Analytics. In *Proc. of the 18th ACM/IFIP/USENIX Middleware Conf. (Middleware '17).* ACM, New York, NY, USA, 185–197. DOI: https://doi.org/10.1145/3135974.3135989

[116] Medhabi Ray, Chuan Lei, and Elke A. Rundensteiner. 2016. Scalable Pattern Sharing on Event Streams. In *Proc. of the 2016 Int. Conf. on Management of Data (SIGMOD '16).* ACM, New York, NY, USA, 495–510. DOI: https://doi.org/10.1145/2882903.2882947

[117] Nicoló Rivetti, Emmanuelle Anceaume, Yann Busnel, Leonardo Querzoni, and Bruno Sericola. 2016. Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems. In *Proc. of the 17th Int. Middleware Conf. (Middleware '16).* ACM, New York, NY, USA, Article 11, 12 pages. DOI: https://doi.org/10.1145/2988336.2988347

[118] Nicoló Rivetti, Leonardo Querzoni, Emmanuelle Anceaume, Yann Busnel, and Bruno Sericola. 2015. Efficient Key Grouping for Near-optimal Load Balancing in Stream Processing Systems. In *Proceedings of the 9th ACM Int. Conf. on Distributed Event-Based Systems (DEBS '15).* ACM, New York, NY, USA, 80–91. DOI: https://doi.org/10.1145/2675743.2771827

[119] S. Rizou, F. Dürr, and K. Rothermel. 2010. Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks. In *2010 Proc. of 19th Int. Conf. on Computer Communications and Networks.* 1–6. DOI: https://doi.org/10.1109/ICCCN.2010.5560127

[120] Omran Saleh, Heiko Betz, and Kai-Uwe Sattler. 2015. Partitioning for Scalable Complex Event Processing on Data Streams. In *New Trends in Database and Information Systems II.* Springer, Cham, 185–197. https://link.springer.com/chapter/10.1007/978-3-319-10518-5_15 DOI: 10.1007/978-3-319-10518-5_15.

[121] Omran Saleh and Kai-Uwe Sattler. 2015. The Pipeflow Approach: Write Once, Run in Different Stream-processing Engines. In *Proceedings of the 9th ACM Int. Conf. on Distributed Event-Based Systems (DEBS '15).* ACM, New York, NY, USA, 368–371. DOI: https://doi.org/10.1145/2675743.2776774

[122] B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. 2011. Esc: Towards an Elastic Stream Computing Platform for the Cloud. In *2011 IEEE 4th Int. Conf. on Cloud Computing.* 348–355. DOI: https://doi.org/10.1109/CLOUD.2011.27

[123] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K. Wu. 2009. Elastic scaling of data parallel operators in stream processing. In *2009 IEEE International Symposium on Parallel Distributed Processing.* 1–12. DOI: https://doi.org/10.1109/IPDPS.2009.5161036

[124] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. 2012. Auto-parallelizing Stateful Distributed Streaming Applications. In *Proc. of the 21st Int. Conf. on Parallel Architectures and Compilation Techniques (PACT '12).* ACM, New York, NY, USA, 53–64. DOI: https://doi.org/10.1145/2370816.2370826

[125] S. Schneider, M. Hirzel, B. Gedik, and K. L. Wu. 2015. Safe Data Parallelism for General Streaming. *IEEE Trans. Comput.* 64, 2 (Feb. 2015), 504–517. DOI: https://doi.org/10.1109/TC.2013.221

[126] Scott Schneider, Joel Wolf, Kirsten Hildrum, Rohit Khandekar, and Kun-Lung Wu. 2016. Dynamic Load Balancing for Ordered Data-Parallel Regions in Distributed Streaming Systems. In *Proc. of the 17th Int. Middleware Conf. (Middleware '16).* ACM, New York, NY, USA, 21:1–21:14. DOI: https://doi.org/10.1145/2988336.2990475

[127] M. A. Shah, J. M. Hellerstein, Sirish Chandrasekaran, and M. J. Franklin. 2003. Flux: an adaptive partitioning operator for continuous query systems. In *Proc. of the 19th Int. Conf. on Data Engineering (Cat. No.03CH37405).* 25–36. DOI: https://doi.org/10.1109/ICDE.2003.1260779

[128] Anatoli Shein, Panos Chrysanthis, and Alexandros Labrinidis. 2018. SlickDeque: High Throughput and Low Latency Incremental Sliding-Window Aggregation. (2018). DOI: https://doi.org/10.5441/002/edbt.2018.35

[129] S. Shevtsov, M. Berekmeri, D. Weyns, and M. Maggio. 2018. Control-Theoretical Software Adaptation: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 44, 8 (Aug 2018), 784–810. DOI: https://doi.org/10.1109/TSE.2017.2704579

[130] A. Shukla and Y. Simmhan. 2018. Toward Reliable and Rapid Elasticity for Streaming Dataflows on Clouds. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS).* 1096–1106. DOI: https://doi.org/10.1109/ICDCS.2018.00109

[131] 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. *Information Sciences* 319 (2015), 92 – 112. DOI: https://doi.org/10.1016/j.ins.2015.03.027 Energy Efficient Data, Services and Memory Management in Big Data Information Systems.

[132] SystemS 2019. IBM System S Project website. (jan 2019). Retrieved 2019-01-15 from http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=2534/

[133] Yuzhe Tang and Bugra Gedik. 2013. Autopipelining for data stream processing. *IEEE Transactions on Parallel and Distributed Systems* 24, 12 (2013), 2344–2354.

[134] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. 2017. Low-Latency Sliding-Window Aggregation in Worst-Case Constant Time. In *Proceedings of the 11th ACM Int. Conf. on Distributed and Event-based Systems (DEBS '17).* ACM, New York, NY, USA, 66–77. DOI: https://doi.org/10.1145/3093742.3093925

[135] Kanat Tangwongsan, Martin Hirzel, Scott Schneider, and Kun-Lung Wu. 2015. General Incremental Sliding-window Aggregation. *Proc. VLDB Endow.* 8, 7 (Feb. 2015), 702–713. DOI: https://doi.org/10.14778/2752939.2752940

[136] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Compiler Construction.* Springer, Berlin, Heidelberg, 179–196. DOI: https://doi.org/10.1007/3-540-45937-5_14

[137] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal. 2014. AJIRA: A Lightweight Distributed Middleware for MapReduce and Stream Processing. In *2014 IEEE 34th Int. Conf. on Distributed Computing Systems.* 545–554. DOI: https://doi.org/10.1109/ICDCS.2014.62

[138] J. S. v. d. Veen, B. v. d. Waaij, E. Lazovik, W. Wijbrandi, and R. J. Meijer. 2015. Dynamically Scaling Apache Storm for the Analysis of Streaming Data. In *2015 IEEE First Int. Conf. on Big Data Computing Service and Applications.* 154–161. DOI: https://doi.org/10.1109/BigDataService.2015.56

[139] Uri Verner, Assaf Schuster, and Mark Silberstein. 2011. Processing Data Streams with Hard Real-time Constraints on Heterogeneous Systems. In *Proceedings of the Int. Conf. on Supercomputing (ICS '11).* ACM, New York, NY, USA, 120–129. DOI: https://doi.org/10.1145/1995896.1995915

[140] Y. H. Wang, K. Cao, and X. M. Zhang. 2013. Complex event processing over distributed probabilistic event streams. *Computers & Mathematics w. Appl.* 66, 10 (Dec. 2013), 1808–1821. DOI: https://doi.org/10.1016/j.camwa.2013.06.032

[141] D. Warneke and O. Kao. 2011. Exploiting Dynamic Resource Allocation for Efficient Parallel Data Processing in the Cloud. *IEEE Transactions on Parallel and Distributed Systems* 22, 6 (June 2011), 985–997. DOI: https://doi.org/10.1109/TPDS.2011.65

[142] Thomas Weigold, Marco Aldinucci, Marco Danelutto, and Vladimir Getov. 2012. Process-driven Biometric Identification by Means of Autonomic Grid Components. *Int. J. Auton. Adapt. Commun. Syst.* 5, 3 (July 2012), 274–291. DOI: https://doi.org/10.1504/IJAACS.2012.047659

[143] 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 *Proc. of the 9th ACM/IFIP/USENIX Int. Conf. on Middleware (Middleware '08).* Springer-Verlag New York, Inc., New York, NY, USA, 306–325. http://dl.acm.org/citation.cfm?id=1496950.1496970

[144] Louis Woods, Jens Teubner, and Gustavo Alonso. 2010. Complex Event Detection at Wire Speed with FPGAs. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 660–669. DOI:https://doi.org/10.14778/1920841.1920926

[145] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance Complex Event Processing over Streams. In *Proc. of the 2006 ACM SIGMOD Int. Conf. on Management of Data (SIGMOD '06).* ACM, New York, NY, USA, 407–418. DOI: https://doi.org/10.1145/1142473.1142520

[146] Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. 2012. Parallelizing stateful operators in a distributed stream processing system: how, should you and how much?. In *Proceedings of the 6th ACM Int. Conf. on Distributed Event-Based Systems.* ACM, 278–289. http://dl.acm.org/citation.cfm?id=2335515

[147] Y. Wu and K. L. Tan. 2015. ChronoStream: Elastic stateful stream computation in the cloud. In *2015 IEEE 31st Int. Conf. on Data Engineering.* 723–734. DOI:https://doi.org/10.1109/ICDE.2015.7113328

[148] Ying Xing, Jeong-Hyon Hwang, Uğur Çetintemel, and Stan Zdonik. 2006. Providing Resiliency to Load Variations in Distributed Stream Processing. In *Proc. of the 32Nd Int. Conf. on Very Large Data Bases (VLDB '06).* VLDB Endowment, 775–786. http://dl.acm.org/citation.cfm?id=1182635.1164194

[149] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. 2014. T-Storm: Traffic-Aware Online Scheduling in Storm. In *2014 IEEE 34th Int. Conf. on Distributed Computing Systems.* 535–544. DOI:https://doi.org/10.1109/ICDCS.2014.61

[150] L. Xu, B. Peng, and I. Gupta. 2016. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *2016 IEEE Int. Conf. on Cloud Engineering (IC2E).* 22–31. DOI:https://doi.org/10.1109/IC2E.2016.38

[151] Keiichi Yasumoto, Hirozumi Yamaguchi, and Hiroshi Shigeno. 2016. Survey of real-time processing technologies of iot data streams. *Journal of Information Processing* 24, 2 (2016), 195–202.

[152] Nikos Zacheilas, Vana Kalogeraki, Nikolas Zygouras, Nikolaos Panagiotou, and Dimitrios Gunopulos. 2015. Elastic complex event processing exploiting prediction. In *Big Data (Big Data), 2015 IEEE Int. Conf. on.* IEEE, 213–222.

[153] Nikos Zacheilas, Nikolas Zygouras, Nikolaos Panagiotou, Vana Kalogeraki, and Dimitrios Gunopulos. 2016. Dynamic Load Balancing Techniques for Distributed Complex Event Processing Systems. In *Distributed Applications and Interoperable Systems.* Springer, Cham, 174–188. https://link.springer.com/chapter/10.1007/978-3-319-39577-7_14 DOI: 10.1007/978-3-319-39577-7_14.

[154] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13).* ACM, New York, NY, USA, 423–438. DOI:https://doi.org/10.1145/2517349.2522737

[155] E. Zeitler and Tore Risch. 2011. Massive Scale-out of Expensive Continuous Queries. In *Proc. of the VLDB Endowment, Vol. 4, No. 11.*

[156] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. 2015. How to Bid the Cloud. In *Proc. of the 2015 ACM Conf. on Special Interest Group on Data Communication (SIGCOMM '15).* ACM, New York, NY, USA, 71–84. DOI:https://doi.org/10.1145/2785956.2787473

[157] D. Zimmer and R. Unland. 1999. On the semantics of complex events in active database management systems. In *Data Engineering, 1999. Proceedings., 15th Int. Conf. on.* 392–399. DOI:https://doi.org/10.1109/ICDE.1999.754955

[158] Nikolas Zygouras, Nikos Zacheilas, Vana Kalogeraki, Dermot Kinane, and Dimitrios Gunopulos. 2015 *(. EDBT '15).* OpenProceedings.org. DOI:https://doi.org/10.5441/002/edbt.2015.65