To Migrate or Not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing

Espen Volnes, Thomas Plagemann, and Vera Goebel

***Abstract*—One of the most important issues in distributed data stream processing systems is using operator migration to handle highly variable workloads cost-efficiently and adapt to the needs at any given time on demand. Operator migration is a complex process involving changes in the state and stream management of a running query, typically without any data loss, and with as little disruption to the execution as possible. This tutorial aims to introduce operator migration, explain the core elements of operator migration, and provide the reader with a good understanding of the design alternatives used in existing solutions. We developed a conceptual model to explain the fundamentals of operator migration and introduce a unified terminology, leading to a taxonomy of existing solutions. The conceptual model separates mechanisms, i.e., how to migrate, and policy, i.e., when to migrate. This separation is further applied to structure the description of existing solutions, offering the reader an algorithmic perspective on various design alternatives. To enhance our understanding of the impact of various design alternatives on migration mechanisms, we also conducted an empirical study that provides quantitative insights. The operator downtime for the naïve migration approach is almost 20 times longer than when applying an incremental checkpoint-based approach.**

***Index Terms*—Middleware, adaptive systems, big data.**

I. Introduction

DISTRIBUTED stream processing (DSP) has been researched for more than 20 years, and is becoming ubiquitous in application domains where real-time decision-making is essential [1], like the Internet of Things (IoT), fraud, and anomaly detection, smart cities [2], and autonomic systems. DSP is a useful technology whenever there is too much data to store all of it, and when the data is only valuable shortly after it is generated. Deep learning can be applied to facilitate analytics on streaming data in IoT [3]. Industry 4.0 is a term that means the fourth generation of the industrial revolution and applies stream processing to do data collection, analysis, storing and querying [4].

Manuscript received 5 September 2022; revised 13 March 2023, 26 June 2023, and 20 September 2023; accepted 30 October 2023. Date of publication 7 November 2023; date of current version 27 February 2024. This work was supported by the Parrot Project (Research Council of Norway, IKTPluss) under Grant 311197. *(Corresponding author: Espen Volnes.)*

Espen Volnes is with the Department of Informatics, The University of Oslo, 0373 Oslo, Norway (e-mail: espenvol@ifi.uio.no).

Thomas Plagemann and Vera Goebel are with the Department of Informatics, The University of Oslo, 0373 Oslo, Norway (e-mail: plageman@ifi.uio.no; goebel@ifi.uio.no).

Digital Object Identifier 10.1109/COMST.2023.3330953

DSP is currently used by companies that need to process and analyze billions of events every day. For example, the popular stream processing engine Apache Flink [5] is used by Alibaba, AWS, Comcast, Ebay, Huawei, Lyft, Uber, Zalando, and many more companies, to perform real-time processing [6]. Another indicator of the wide use and importance of stream processing is that most cloud vendors offer support for deploying managed stream processing pipelines [7], and a sign of its future relevance is the estimated economic impact of the IoT industry, estimated to be between \$3.9 trillion and \$11.1 trillion a year by 2025, around 11% of the global economy [8].

Stream processing engines (SPE) come in several flavors, are deployed in different environments (i.e., cloud, fog, edge, in-network), and perform data stream management, real-time stream analytics, event stream processing, and complex event processing (CEP). The common denominator in all these systems is that data arrive continuously (generally as tuples) from multiple sources, and need to be processed as soon as they arrive (in memory) to enable immediate decision-making. Thus, the response time must be short, even in case of large loads.

SPEs take queries as input and compile them into operator graphs. In the simple example in Figure 1 the query at the top of the figure is compiled to an operator graph with two data producing operators (Auction and Bid stream), a join operator, and a data consuming operator (Section II-C builds on this simple example and gives further details). Operator graphs are directed acyclic graphs (DAGs), as illustrated in Figure 1, that represent the logical execution of a query, which includes the operators (i.e., state management of subqueries) and the dependencies between them (i.e., stream management) represented as vertices. If these operators are mapped to several physical hosts and form an overlay network, a DSP system is established. Incoming data tuples to an operator are processed, e.g., by filtering and joining as in Figure 1, or transforming, aggregating, or running a user-defined function.

A key requirement for DSP is the ability to handle system dynamics, like changes in workload, resource availability, and mobility. Operator migration is the key mechanism for handling such changes. The four primary goals that motivate different operator migration solutions are: (1) to re-balance uneven distribution of computational tasks across nodes (load balancing), (2) adapting the amount of allocated resources to increasing or decreasing workload (elasticity), (3) maintaining system operability even in the presence of hardware failure

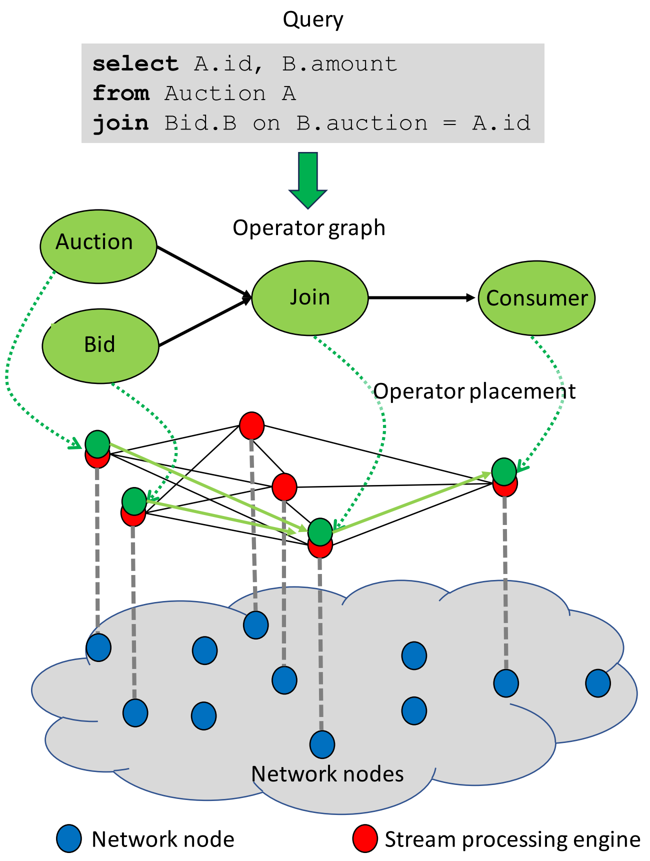


Fig. 1. Overview of operator placement.

and other faults (fault tolerance), and (4) maintaining or optimizing the Quality of Service (QoS). Operator migration entails (1) state management to move the state of the operator from an old host to a new host, and (2) stream management to change data stream routing in the overlay network. Decisions on when to migrate the data and where to migrate them to are key aspects of operator migration. The potential approaches to state management, stream management, and decision-making as well as their combinations result in a large design space for operator migration algorithms.

This tutorial aims to give the reader a good understanding of (1) the need for operator migration, (2) the core elements of operator migration, (3) the design of existing solutions, and (4) how design decisions can impact the performance of operator migration solutions. To this end, we develop a conceptual model that captures the fundamental components of operator migration, i.e., the components on which all solutions are based and their relationships. This model provides a unified terminology and is used to establish a taxonomy of existing solutions. Based on this, we describe the main existing operator migration solutions.

Operator migration introduces some form of cost, like freeze time during migration or increased resource consumption to move the state of the operator. Keeping these costs low is a core requirement in the design of operator migration algorithms. Furthermore, during decision-making, it is important to balance the costs of migration against its benefits. There is a general awareness of this trade-off, but surprisingly, few studies have explicitly described how costs and benefits are considered in the migration algorithm and decision-making. Therefore, we place particular emphasis on costs and benefits in our analysis of work in this area. We structure the description of existing solutions into two parts:

- Mechanisms: How does the operator migration work, and which mechanisms are used?
- Policies: When should operator migration be performed, and how is the migration decision executed?

In addition to this functional view of operator migration, we perform an empirical study to gain and mediate quantitative insights into different operator migration and decision models. The aim is to illustrate the quantitative effect of different design decisions. This empirical quantification demonstrates the advantage of a comprehensive migration model beyond the contribution of the literature. We use Apache Flink [5] and Siddhi [9], two operator migration algorithms, and apply part of the NEXMark benchmark [10] as workload to measure the run-time performance.

A. *Tutorial Novelty and Contributions*

To the best of our knowledge, this tutorial represents the first comprehensive effort to explain operator migration mechanisms and related decision-making in data stream processing systems. There exists a short description of a tutorial given in 2014 [11] by Heinze et al. However, due to space limitations the published version of the tutorial cannot be comprehensive and it can not capture developments after 2014. Therefore, this tutorial is unique in its scope and contribution to the current state of knowledge on the topic.

There is a range of surveys that cover operator migration [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] to a certain extent. However, all these surveys have a broader scope than this tutorial and do therefore not explore operator migration in corresponding depth and detail. For example, none of these surveys presents a type of framework for operator migration like a taxonomy or a conceptual model, and none of the surveys provides a unified terminology for operator migration. This tutorial distinguishes itself by presenting different types of operator migration algorithms in detail and the relationship between the cost and benefit of migration, the decision to migrate, and the migration algorithm. Furthermore, the surveys do not give the reader an insight into the quantitative impact of certain design decisions for operator migration.

Table I characterizes related surveys with respect to their focus area as well as the questions of whether:

1) any kind of framework, like a taxonomy or a conceptual model for operator migration is provided;
2) details of the decision-making process are given, such as the goal of migration, the performance of migration, and the cost of migration;
3) the survey methodology;
4) the deployment environment is considered in the discussion of existing solutions;
5) the paper uses some experimental investigations to demonstrate and quantify the effect of different design decisions.

TABLE I
SUMMARY OF RELATED SURVEYS AND COMPARISON WITH THIS TUTORIAL (T)

Papers	Focus area	Framework for operator migration	Details of migration decision	Methodology	Deployment	Experiment
[12]	Placement for Internet-Scale stream processing	No	Partially	Explanatory	Yes	No
[13]	Elastic stream processing in the cloud	No	No	Enumerative & explanatory	No	No
[14]	Stream processing optimizations	No	No	Enumerative & explanatory	No	Yes
[16]	State management in big-data processing systems	No	No	Enumerative & explanatory	No	No
[18]	Adaptation of stream processing	No	No	Enumerative	No	No
[17]	Parallelization and elasticity in stream processing	No	Yes	Enumerative & explanatory	Yes	No
[19]	Resource management and scheduling in stream processing	No	Yes	Enumerative & explanatory	Yes	No
[20]	Geo-distributed big-data analytics	No	No	Enumerative & explanatory	Yes	No
[21]	Runtime adaptation of stream processing	No	Yes	Enumerative & explanatory	Yes	No
[22]	Self-adaptation on parallel stream processing	No	No	Enumerative	Yes	No
T	Operator Migration in stream processing	Yes	Yes	Enumerative & explanatory	Yes	Yes

Lakshmanan et al. [12] focused on operator placement and reconfigurations for Internet-scale data stream systems. They distinguished between reconfiguration solutions based on where the change is made: either in the network, data, or flow graph. Moreover, different triggers for migrations were studied, such as thresholds, constraint violations, and periodic re-evaluations. However, they did not investigate the different varieties of operator migration in any detail. Hummer et al. [13] focused on elasticity in the cloud, which can be achieved through event reordering and prioritization, load shedding, deferred processing, and operator migration. They also investigated the state in different types of windows and how these have to be migrated in case of a scaling operation. Moreover, the cost of migration is also problematized, i.e., that performing a scaling operation costs time that might adversely affect the performance of the system. Similarly, Röger and Mayer [17] investigated elasticity and parallelization in stream processing. Operator migration is in this context one method to achieve elasticity, but details about operator migration mechanisms and migration decision-making are not given.

Hirzel et al. [14] cataloged different types of stream processing optimizations, including operator graph optimizations, operator placement, load balancing, state sharing, batching, load shedding, and several more. Operator migration is relevant for load balancing and operator placement, but the paper's aim is too broad to describe migration in the detail targeted in this tutorial. Microbenchmarks with InfoSphere Streams are used to demonstrate the profitability of the optimization, but operator migration is not experimentally investigated.

To et al. [16] studied state management in stream processing systems, with a focus on big data cloud-based systems. They investigated existing ways of representing state in the system, optimizing performance, and provide insights into various state management techniques. Operator migration, elasticity and load balancing are three of the 18 concepts of state management that are presented.

Similarly, de Assunção et al. [15] explored migration in relation to stream processing and edge computing. They provided informative summaries of multiple generations of DSP systems and analyzed existing work on elasticity to adapt resource allocation to handle the workload of stream processing services. Operator migration is one of many means for elasticity and the inner workings of operator migration are not analyzed and described in detail.

Liu and Buyya [19] presented a taxonomy of resource management and scheduling in DSP. Operator migration and state management are not explored in the paper, as it is assumed that the mechanisms have been studied and are provided by the state-of-the-art systems. On the other hand, the decision-making process is investigated in depth.

Qin et al. [18] defined a taxonomy for different live reconfigurations in SPEs. This includes 17 types of adaptations, including operator migration, load balancing, and scaling. However, this tutorial investigates these three issues as fundamentally being similar types of adaptations. Furthermore, the survey [18] is of pure enumerative nature and does not aim to explain how operator migration works. Similarly, Cardellini et al. [21] presented a survey on run-time adaptations. They studied the methodological and architectural approaches for adaptation control and differentiate 14 adaptation mechanisms. Their presentation of adaptation goals includes a popularity analysis of metrics used for adaptation. In Section V of this tutorial, the metrics used by papers is discussed in depth. Bergui et al. [20] surveyed geo-distributed frameworks, some of which are described in this tutorial. Moreover, they discussed several challenges pertaining to geo-distributed data analytics, where operator migration plays only a minor role in some of the solutions.

Vogel et al. [22] presented a systematic literature survey of self-adaptation mechanisms of parallel stream processing. Operator migration is not explored in this paper, but a conceptual framework is proposed that includes adaptation goals and decision-making. The scope is much broader than this tutorial, and can therefore not go into the same depth in the related topics.

The main contributions of this tutorial are as follows:

- We propose a conceptual model of operator migration that provides a unified terminology and leads to a taxonomy of operator migration. Moreover, this model facilitates the development of new operator migration solutions.
- We describe the main works on operator migration and analyze not only current stream management and state management solutions (i.e., mechanisms), but also emphasize a cost-benefit analysis of the migration decision (i.e., policies).
- We perform an experimental study involving two migration algorithms on Apache Flink and Siddhi to gain insight into the quantitative aspects of operator migration.

B. Literature Search Methodology

The conceptual model of operator migration has been created in an iterative manner using the existing works in the literature. The focus has been to select the works that describe their migration mechanism in detail, or how the migration decisions are made.

We have searched for existing literature in the most popular search engines, e.g., Google Scholar and Web of Science. The searches have included DSP and many keywords that relate to operator migration, elasticity, load balancing and fault tolerance. We have included works in the tutorial that have a substantial contribution to migration mechanisms or migration decision-making. This search is not straightforward, since sometimes, operator migration might be applied in a way where it is not the main contribution. Moreover, while some works categorize operator migration as a specific subset of big data adaptation techniques [18], our tutorial takes a broader perspective. It presents operator migration not as a specific adaptation, but as a crucial mechanism that enables other key features of big data adaptation, including load balancing, elasticity, QoS and fault tolerance.

Many works exist on migration of services in Multi-access Edge Computing (MEC) [23], [24], [25], [26], [27], [27], [28], [29], [30], [31], [32], [33], [34], [35], [36], [37], [38], [39]. One of the core features of MEC is the ability to offload heavy tasks to a host with more resources or better conditions for completing the task [40]. The entities to migrate may be virtual machines (VM) [24], containers [30], [37] or Virtual Network Functions [27]. The methods proposed in these papers may also be applicable to operator migration when it comes to deciding when and where to migrate, which is discussed in Section V. However, the migration mechanisms that have been developed specifically for DSP and described in Section IV, are different from the ones used when migrating services, since operators in DSP are more fine-grained. The migration entity is not an entire application, but rather an internal state. Whereas the entire service must be at the new host until the service can be restarted in the case of MEC, certain optimizations can be made in DSP, depending on what type of stateful operator is migrated.

Table II lists the studies considered in this work that form the foundation of the conceptual model. It classifies them according to the environment of their deployment and the goal of migration, which are important factors for the migration decision and placement. The most common deployment environments for DSP are cloud, fog, and edge networks. *Cloud* has been used to classify data center applications that might handle very high throughputs, and can scale the systems both horizontally and vertically to handle variable traffic loads. The pay-as-you-go business model makes the hardware provisioning easier [41]. The concepts of fog and edge are relatively new terms that seem similar, but have some significant differences. *Edge* computing often focuses on offloading heavy tasks from local resource-constrained devices to either a close base station or a data center [42]. With edge computing, heavy tasks, like deep learning computation and videogames, can be executed using edge devices such as smartphones and laptops [43]. *Fog* is an extension of the cloud in which the computing tasks of an application are distributed on multiple devices, including end devices, edge resources, and the cloud itself [44], [45]. As such, clients may send most information to a server close to them instead of a centralized data center to reduce energy consumption, congestion on the Internet, and response times for clients.

Table III lists the goals of migration and the overlap between studies in the area in terms of percentage. For instance, 40% of the studied papers on elasticity also consider load balancing. This is a common combination, because load balancing can be used after performing a scaling operation to redistribute the load. Few fault tolerance-based solutions describe migration mechanisms, but it is natural that fault tolerance overlaps with load balancing or elasticity as they are often cloud-based solutions, and steps to restore the number of states of a node are similar to those of a scale-in operation. Approaches that use QoS constraints on operators to determine when to migrate, often also use load balancing. This is because a clear sign that workload rebalancing is necessary is when the QoS guarantees of an operator have been violated.

C. Tutorial Structure

Figure 2 sketches the structure of this tutorial, i.e., the sections and some of their content, and identifies the three core parts of the tutorial. Section II describes some basic concepts of distributed data stream processing. The first core part introduces in Section III a conceptual framework for operator migration. The two main concerns of operator migration, i.e., to move the operator state from one host to another host and to decide whether to migrate, structure the conceptual model as well as the other two core parts of the tutorial. Sections IV and V form the second part of the tutorial. The aim of this part is to explain to the reader how the concepts are applied in existing research works to design operator migration algorithms (Section IV) and to perform migration decisions (Section V). The third part, i.e., Section VI, follows a "hands-on" approach and aims to give the reader quantitative insights gained through empirical investigations. On the one hand, decision models for operator migrations are developed and

TABLE II
OVERVIEW OF STUDIES ON THE CATEGORIES OF OPERATOR MIGRATION

Category	Sub-category	Papers
Deployment environment	Cloud	[46–82]
	Fog	[83–92]
	Edge	[93–102]
Migration goal	Load balancing	[15, 46, 47, 50, 52, 53, 57, 58, 60, 62–64, 66, 69, 70, 73, 76–78, 85–87, 93, 102–111]
	Elasticity	[50, 53, 55, 60, 61, 63, 65, 67, 68, 78, 80, 81, 89, 91, 92, 101, 103–105, 107, 110–116]
	Fault tolerance	[48, 59, 76, 78, 94, 101, 113]
	QoS	[49, 67, 68, 71, 72, 74, 78, 84, 86–88, 90, 92, 95, 96, 98, 99, 104, 106, 107, 117, 118]

TABLE III
GOALS OF MIGRATION AND THE OVERLAP IN STUDIES IN THE AREA

Migration goal	Load balancing	Elasticity	QoS	Fault tolerance
Load balancing	100% (33/33)	33% (11/33)	18% (6/33)	6% (2/33)
Elasticity	40% (11/27)	100% (27/27)	22% (6/27)	11% (3/27)
Fault tolerance	28% (2/7)	42% (3/7)	14% (1/7)	100% (7/7)
QoS	27% (6/22)	27% (6/22)	100% (22/22)	4% (1/22)

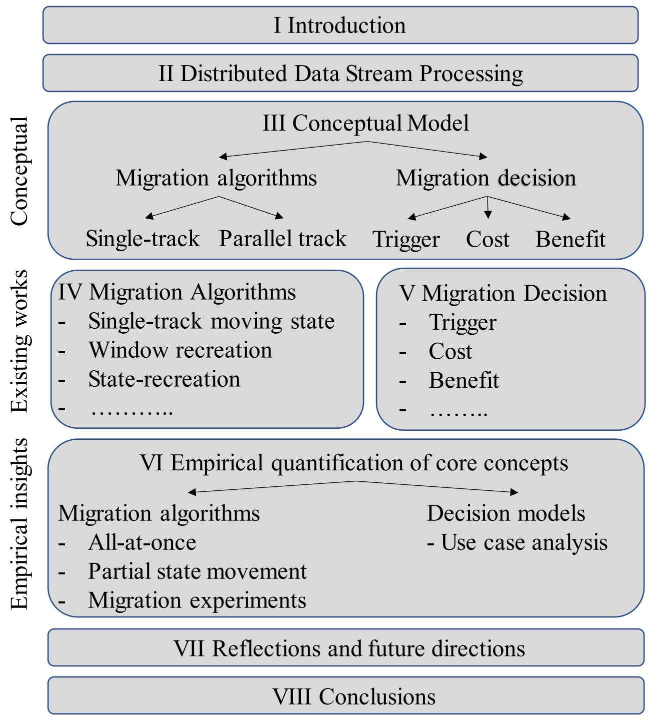


Fig. 2. Paper structure.

TABLE IV
GLOSSARY OF TERMS

Term	**Definition**
Data stream	A continuous flow of data, typically consisting of a sequence of data tuples that can be processed in a streaming fashion, without the need to store everything in memory.
Downstream node	A node that receives data tuples from other nodes in a stream processing topology.
Elasticity	The ability to adaptively scale computational resources up, down, out or in, according to real-time needs.
Load balancing	Distributing computational tasks evenly across available nodes to avoid bottlenecks and maximize resource usage.
Migration trigger	An event or set of conditions that triggers operator migration.
Operator state	The temporary storage of historical data and intermediate results by an operator, essential for producing accurate processing outcomes.
Proactive migration	Triggering operator migration in anticipation of potential issues or changes in system requirements, rather than in response to them.
QoS (Quality of Service)	A set of performance metrics, such as latency, bandwidth, and availability, that the system aims to optimize.
Reactive migration	Triggering operator migration in response to current system conditions, such as failures or performance degradation.
SPE operator	A computational element that processes data streams, e.g., transforming, filtering, aggregating or joining the data.
Tuple latency	The time it takes for a data tuple to be processed or moved through the system.
Upstream node	A node that sends data tuples to other nodes in a stream processing topology.

analyzed through a use-case study, and on the other hand, two different operator migration algorithms are implemented, and their performance is analyzed through experimentation.

To ensure that readers from various backgrounds have a clear and unified understanding of the key terms used in this tutorial, we provide a glossary of critical terms in Table IV. These terms are integral to the discussion and understanding of migration algorithms, state management, and other aspects covered in this paper. The glossary covers terminology that is prevalent in the context of DSP systems, offering definitions aimed at beginners in the field. Readers are encouraged to familiarize themselves with these terms for a comprehensive understanding of the subsequent sections.

In summary, the tutorial first introduces a conceptual model, presents and discusses afterwards design choices of existing works, and demonstrates in the last part the impact of particular design choices through empirical studies. The tutorial is completed through some reflections and discussion of future directions in Section VII and the conclusions in Section VIII.

II. DISTRIBUTED DATA STREAM PROCESSING

A. Data Stream Processing

Data stream processing deals with continuous processing of data tuples. The system model applied in this tutorial is based on [51].

A data stream is an unbounded sequence of tuples that are continuously generated over time. It is denoted as $S = t_1, t_2, t_3, \ldots,$ where t_i represents the i_{th} tuple in the stream. For example, a data stream that represents stock market prices could be denoted as $S = t_1, t_2, t_3, \ldots,$ where $t_i = (\text{symbol, AAPL}),$ $(\text{price, } 150.23)$, $(\text{time, } 2022\text{-}02\text{-}14\ 10{:}30{:}00)$, representing the stock symbol, price, and timestamp of the i_{th} price update in the stream.

A tuple is an ordered list of attribute-value pairs that represents a single unit of data. It is denoted as a set of key-value pairs, where the keys represent the attribute names and the values represent the corresponding attribute values. For example, a tuple that represents a person's information could be denoted as (name, John), (age, 30), (gender, male). Instead of including the attribute names in the tuples, a data stream expects the incoming tuples to follow a schema, and thus, the attributes are inferred.

A query is a function that processes a data stream and returns a result based on some criteria. It is denoted as Q(S), where S is the input data stream and Q is the function that defines the query. For example, a query that computes the average of a stream of numbers could be denoted as $Q(S) = (1/n) * \sum n_i = 1x_i$, where n is the number of elements in the stream, x_i is the i_{th} element in the stream, and $\sum$ is the summation operator.

Nodes in an operator network can be static or mobile, and have one or more of the following roles:

- *Data producer:* Examples of this include sensors that convert analog signals into data tuples, often with a fixed sampling rate, and software monitors that might create data tuples at a dynamic rate. Crucially, the operator network must be able to process all tuples produced by these sources for further processing.
- *Data consumer:* These are nodes that request a service, and typically have some QoS requirement, such as less than a given tuple latency.
- *Operator host:* These nodes execute at least one operator and contribute to event forwarding in the operator network, i.e., map the input events (from upstream nodes) of the operators they execute to output events, and forward them to downstream nodes in the operator network.

Data stream processing queries may be *stateful* or *stateless*. The focus of this paper is on the stateful queries. For instance, when joining two data streams, one tuple arrives before the other, and is placed within a data window, where it will remain until it expires and is deleted. When aggregating state, such as counting words, we are typically interested in creating an aggregate per group/key. In concurrent systems, each key produces output separate from other keys, and as such, these aggregates can be produced by different threads/processes. Therefore, it is common to parallelize such queries, and execute some keys on one host and other keys on another host, in a cluster.

Numerous data stream processing systems exist, including but not limited to Storm [119], Flink [5], Esper [120], and Siddhi [9]. For a more comprehensive list, please refer to the survey by Isah et al. [121]. In Section VI, we run real-world experiments with Siddhi and Flink. As these systems are difficult to execute, there have been efforts to simplify the interface for running such systems. Apache Beam [122] is a system that provides a unified interface to existing stream processing systems, where each supported system needs a runner that represents the integration between Beam and a given system. Expose [123] is a stream processing evaluation framework that provides an easy interface for running distributed experiments with any distributed stream processing system that has a wrapper, that implements an API that represents the core functionality of the system. There is also an interest in simulating these systems, and efforts have been made in DCEP-Sim [124] and ECSNeT++ [125].

When it comes to simulation of fog and edge computing that model more generic services, with mobility and service migration, there has been a range of simulators, including, CloudSim [126], iFogSim [127], iFogSim2 [128], EdgeCloudSim [129], FogNetSim++ [130], IoTSim-Edge [131], MobFogSim [132], YAFS [133], PureEdgeSim [134], IoTNetSim [135], SatEdgeSim [136], and IoTSim-Osmosis [137]. These simulators, however, do not focus on data stream processing, and are often more focused on the interactions between edge, fog, and cloud nodes, and using generic services, instead of specific operators, which are necessary in data stream processing.

B. Initial placement

A DSP can be considered to be a set of collaborating SPEs that form an overlay network to process queries over data streams. Consider Figure 1 as an example of such an application. SPEs run on network nodes that provide the computational and networking resources for the DSP overlay. The objective of the initial operator placement is to distribute the processing of a query over network nodes such that the goals of the system can be met as adequately as possible [138]. The first step is to transform a query into an operator graph. An operator graph can be modeled as a DAG in which the operators derived from the query are represented as vertices. The placement of these operators in a network, i.e., finding appropriate network nodes to host the operators, is typically driven by an objective function. Such an objective function typically includes (contradicting) criteria of optimization, like low latency of event delivery, low resource consumption (e.g., bandwidth and energy), reliability, and fault tolerance. Typically, a placement function is used to calculate a placement score based on the criteria of optimization. To find the optimal placement is usually an NP-hard problem [138], and heuristics are often used to find close to optimal solutions. Both centralized and decentralized versions of operator placement can be used to establish an operator network, and are generally implemented as an overlay for the DSP.

DSP is performed in a dynamic context involving variable workload, resource availability, and possibly mobility. As such, initial placement might, after some time, become sub-optimal and the operator network should be adapted by migrating one or several operators to a new host.

TABLE V
OUTPUT WITH STATEFUL MIGRATION

Stream	Attribute 1	Attribute 2
Auction	id: 1	name: Mona Lisa
Auction	id: 2	name: The Scream
Auction	id: 3	name: Salvator Mundi
Auction	id: 4	name: Orange Marilyn

TABLE VI
INPUT AFTER MIGRATION

Stream	Attribute 1	Attribute 2	Attribute 3
Bid	Auction: 2	amount: 105M	bidder: 7
Auction	id: 5	name: Spring	
Bid	Auction: 3	amount: 400M	bidder: 4
Bid	Auction: 1	amount: 800M	bidder: 12
Bid	Auction: 4	amount: 190M	bidder: 1
Bid	Auction: 5	amount: 70M	bidder: 4

TABLE VII
OUTPUT WITH STATELESS MIGRATION

Stream	Attribute 1	Attribute 2
Output	Auction: 6	bid: 70M

TABLE VIII
OUTPUT WITH STATEFUL MIGRATION

Stream	Attribute 1	Attribute 2
Output	Auction: 2	Bid: 105M
Output	Auction: 3	Bid: 400M
Output	Auction: 1	Bid: 800M
Output	Auction: 4	Bid: 190M
Output	Auction: 5	Bid: 70M

C. Naïve Migration Example

To illustrate the challenges of state migration, we go through the simple example introduced in Figure 1. Consider a data stream processing application that involves three entities: the data producers, the operator hosts, and the data consumers. The data producers may include sensor nodes that produce data. This data is sent to the operator hosts to process the tuples, i.e., to transform, filter, aggregate, and join the data. Data with a specific schema is called a data stream. Different data streams have different attributes in them.

In this simple naïve migration example, a node gets overloaded and has to migrate stateful operators to another node. We use a join query that joins all Bid events with their corresponding Auction event. This query is based on the NEXMark benchmark [10], and is also applied in Section VI for the empirical quantification of the conceptual model. As Auction tuples arrive, they are stored as part of the state in the join operator until expiration. In this way, incoming Bid tuples can be matched against Auction tuples.

In Table V, four Auction tuples are described that are sent before the migration starts. Table VI lists the Auction and Bid tuples that are sent after the migration. If no state is migrated, the output will only be one tuple, shown in Table VII. If the Auction tuples from before the migration are migrated, the output will be VIII. The reason why so many more tuples are produced after the migration when Auction tuples are migrated is that the incoming Bid tuples find a matching Auction tuple to join with. Without them being migrated, only the new Auction tuple with id 5 can be joined with. This is a simple example that shows the necessity of state migration. The migration mechanisms that are introduced in Section III-A, and described more extensively in Section IV, face the same problem of making the operator state available on the new host, such that the operators produce the correct output.

III. A CONCEPTUAL MODEL OF OPERATOR MIGRATION

We establish a conceptual model of operator migration to capture the basic concepts and elements on which consensus has been achieved in the literature, and form a unified terminology for operator migration. For the understanding and the design of operator placement, it is important to separate between mechanism and policy. There are two major concerns for operator migration mechanisms: (1) stream management to stop, buffer, redirect, and start streams; and (2) state management to establish the current state of the operator at the new operator host, which may require moving the state from the old host to the new one, and starting a replica for the operator on the new host before the state transfer is finished. All algorithms require some stream management functions, such as stop, start, buffer, and redirect. State management stands apart due to the multitude of design alternatives it presents. For example, one could choose to move the entire state at once or incrementally. Furthermore, it is possible to execute the operator exclusively at one host during migration, or to do so in parallel on multiple hosts. The policy is implemented in the operator migration decision component that needs to determine whether and when to migrate an operator. This involves several steps. Migration decisions first require a trigger for when to make a migration decision and then a placement mechanism to determine the placement that yields the best performance. The cost of the migration must be weighed against its benefit. The policy must determine the degree to which the migration decision should be proactive (e.g., before a host becomes overloaded) or reactive (e.g., when a host is overloaded). The more proactive a migration decision is, the higher uncertainty it has. The more reactive a decision is, the higher cost of migration it has.

Figure 3 illustrates the concepts and building blocks that make up migration algorithms. It also highlights the relevant decision-making processes, outlines the properties of state management mechanisms in migration, and presents the associated costs of migration. Migration cost is important for migration mechanisms and migration decisions, because every migration mechanism introduces some form of costs and the migration decision needs to take the costs into account to determine whether it is worthwhile to migrate. This relationship between the migration mechanism and the migration decision makes the migration cost a core element of the conceptual model. The state management node describes the dimensions of migration mechanisms that are explored in this tutorial.

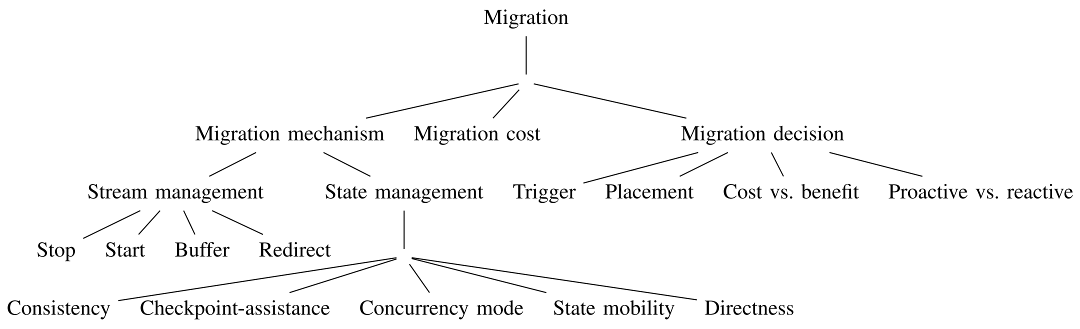


Fig. 3. Concepts of migration.

The remaining structure of this section directly reflects the structure of Figure 3, i.e., a detailed discussion of the components and design alternatives for migration mechanisms is given in Section III-A, followed by an overview of the common cost parameters in Section III-B, and, in Section III-C, the migration decision.

A. Migration Mechanism

The two major concerns of migration mechanisms are state management and stream management. State management is relevant for operators that derive their output based on multiple tuples, e.g., looking for a sequence of tuples using CEP, joining streams, or aggregating tuples over windows [16]. The *state* can be thought of as tuples. The internal state of the operator is, in practice, typically optimized to include only the necessary information for the given operator, such as the given aggregate value for the extent of a window, or as a finite state machine in CEP. In addition to such stateful operators, there exist also stateless operators, like filter and map. These operators do not require state because they process each input tuple independently. Therefore, operator migration distinguishes itself markedly from VM migration, where the entire VM must be transferred to the new host. While some solutions to operator migration, such as the MCEP [98], do include VM migration, VM migration is not addressed in this tutorial. The simplest method of operator migration for a stateful operator is to move it to the new host and replay all necessary historical tuples from the upstream nodes [139]. This technique is used in current publish-subscribe systems, such as Kafka [140], to achieve fault tolerance in stream processing systems like Flink [5]. Using this technique also makes it possible to migrate the data to a different stream processing system, which is usually not possible when extracting the state from the system, because the internal state is system specific. However, as the state can become very large, it is often undesirable to replay all tuples. Therefore, this tutorial focuses on operator migration techniques that extract the state from the stream processing system and move it to the new host.

The purpose of state management in operator migration is to establish, at the new host, an operator with the state of the operator at the old host when switching the processing from the old host to the new host. In a *moving state* algorithm, the old host extracts the state of the operator and sends it to the new host. Some algorithms do not need to perform this task, either because they manage stateless operators, e.g., filter operators, or because the old and the new host can schedule a seamless handover of the operator. In a *parallel-track* algorithm, originally a term used by Zhu et al. [141], both the old and new hosts receive the same tuples for some time during migration. The handover from the old to the new host is carried out gradually such that the downtime of the operator is minimized. The cost of this approach is that upstream nodes must send twice as many tuples during some part of the migration. A parallel-track algorithm with moving state is called *state-recreation*, and one without moving state is called *window-recreation*. These terms are inspired by StreamCloud [103]. In a *single-track* algorithm, the upstream nodes send tuples either to the old host or to the new host.

Stream management deals with notifying upstream and downstream nodes of changes made to the DSP overlay. Typically, nodes have to update their routing table to reflect the new topology at the upstream node, and this results in a redirection of the outgoing stream to the new operator host. To prevent tuples from getting lost when the operator is down, streams might be stopped and tuples need to be buffered. There are three locations at which tuples can be buffered: upstream nodes, the old host, and the new host. The tasks of redirecting streams, stopping streams, buffering streams, and restarting streams are coordinated among the hosts involved through control messages. Both centralized and decentralized coordination is possible. As such, there are several design options that can be implemented for a particular operator migration solution.

For presentation purposes, the taxonomy is divided into two parts: single-track algorithms (Figure 4) and parallel-track algorithms (Figure 5). In single-track algorithms, each tuple is processed either on the old host or the new host, but not both, at any given time. This means that even if an operator is active on both hosts during the migration, each individual tuple is processed exclusively on one host or the other. In contrast, parallel-track algorithms involve tuples being processed on both the old and new hosts simultaneously during the migration period, meaning the same tuple is processed on both hosts. The small text in brackets under some of the categories denotes a term for the given type of algorithm. For instance, a *pause-drain-resume* algorithm is a single-track algorithm without moving state, and a parallel-track algorithm with moving state is a state-recreation algorithm. The most basic operator migration algorithm is a pause-drain-resume

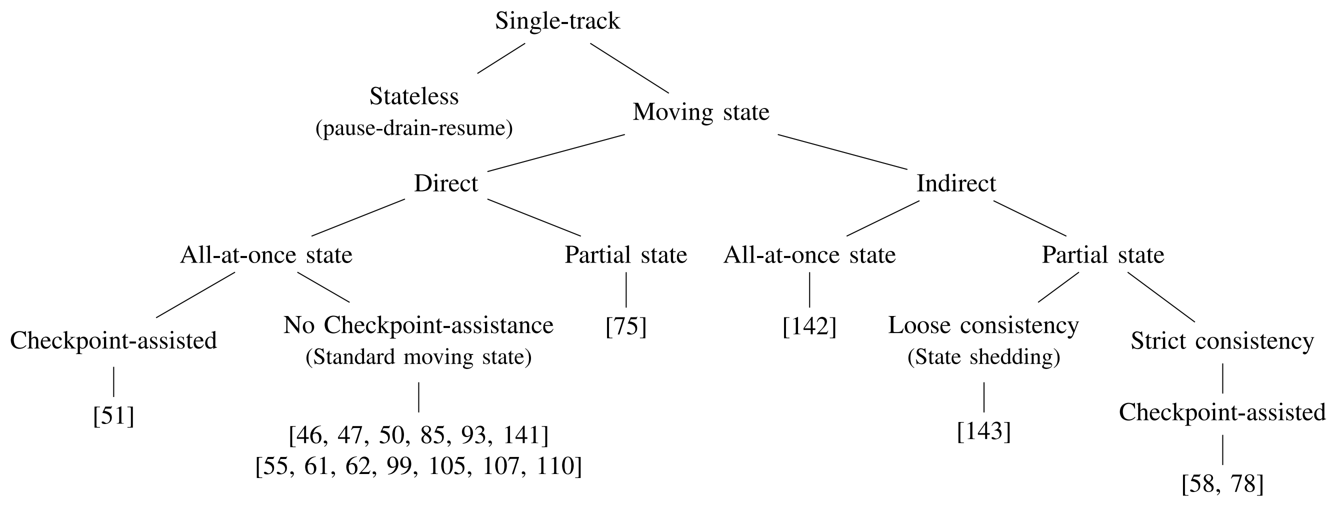


Fig. 4. Single-track migration algorithms.

algorithm, and works only with stateless operators or in cases where some state inconsistency is permitted. The operator to migrate is first started on the new host while the old host is also running it. Then upstream nodes redirect their output streams from the old to the new host. After this, the old host can stop the execution of the operator. Since no state needs to be moved, migration occurs without any downtime. A few control messages must be sent (1) from the controller to the old host, (2) from the old host to the new host, and (3) from the old host to the upstream nodes. As there is no downtime for the operator, any delay caused by these messages is negligible.

When state must be moved and a single-track is used, the operator has some downtime. Specifically, tuples can neither be processed on the old nor the new host while the state is in transmission. In this process, tuples from the upstream nodes must be buffered before the new host can process them. The buffering can be carried out on the upstream nodes, the old host, or the new host. In many cases, tuples can be received after query processing has been stopped. These tuples need to be forwarded from the old host to the new host.

Partial state movement involves splitting the state to be migrated into several parts and moving these parts to the new host while the operator is still processing on the old host. This approach avoids having to stop operator processing for the entire state transfer. If the state is periodically checkpointed and distributed on different nodes, this is called *checkpoint-assisted migration*, and can substantially reduce or eliminate the downtime of the operator. Either the entire state already exists on the new host or an incremental checkpoint is extracted before the operator shuts down, and then sent to the new host. While the last checkpoint is sent to the new host, the operator stops for a much shorter time compared to sending the entire checkpoint at once. A single-track moving state solution can never avoid downtime. This is the reason why parallel-track solutions have been developed.

Parallel-track algorithms differ from single-track algorithms in a fundamental way. They can achieve zero downtime, but at the cost of running the old and new hosts with duplicate input streams and, sometimes, duplicate output streams [113]. A moving-state parallel-track algorithm performs state-recreation, which means that the new host receives the state from the old host while also receiving the same tuples as it does from upstream. A parallel-track algorithm without state migration performs window-recreation, which means that the new host receives the same tuples as the old host until the old tuples expire and they both have the same tuples in their windows. At this point, the upstream nodes redirect their streams to the new host, and it takes over without the tuples being buffered or any waiting time.

Most of the existing works assume fully consistent state for the operator migration. This means that before and after the migration, the internal state of the operators looks like it would if there was no migration, except that some state might arrive in different order. No state is lost. This is an important principle for adaptive stream processing systems; that adaptations occur transparently to the data producers and consumers. In a recent work [143], however, state shedding is presented as the idea of performing a migration where the most important partial states are migrated, and some less important partial states are dropped if the total state is too big to migrate. This is meant to be used in volatile cases where the system fails unless an adaptation is done quickly.

An important motivation for establishing the terminology and building blocks in Figure 4 and 5 is that existing work has described the same concepts by different names. For instance, what Zhu et al. [141] called *parallel-track* is described as *window-recreation* in StreamCloud [103], *smooth migration* in Enorm [62], and the *seamless minimal state* in TCEP [99]. In StreamCloud, a different algorithm called *state-recreation* is also parallel-track, but also involves moving state. In contrast to parallel-track, single-track with moving state is called *disruptive migration* in Enorm [62] and *Pause & Resume* in [11] because it leads to downtime, as opposed to smooth migration that eliminates downtime. Instant migration [62] is single-track migration without moving state. Checkpoint-assisted algorithms have been described in [58], [78], [113]. A characteristic of these algorithms is that a minimal state needs to be sent during migration. It should be noted that what is

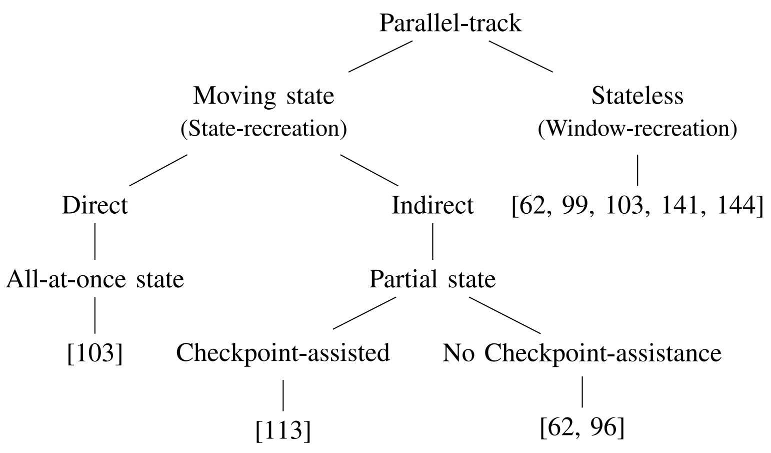


Fig. 5. Parallel-track migration algorithms.

considered state differs among different systems. Therefore, what is considered partial state or all-at-once state might differ. TCEP has a fine-grained migration algorithm that moves during operator migration the entire tuple state all at once, but since the entire operator consists of multiple elements where the tuple state is just part of it, it is not considered a partial state movement algorithm.

B. Migration Cost

Operations performed as part of state management and stream management lead to two classes of the cost of migration, related to resource consumption and temporal aspects. The temporal costs are caused by the fact that the operator is not operational during state extraction, state serialization, state movement, state deserialization, and runtime initialization. The bandwidth required to move the state from the old to the new host is the most commonly considered resource in resource-aware geo-distributed cases [20]. The computational requirements of extracting the state from the old host and the messages needed to coordinate stream management are more commonly considered in centralized data centers. Stream management messages may also have an impact on the operator downtime, e.g., streams from upstream nodes need to be stopped and no events should arrive at the operator until they have been redirected and started again. The operator is further suspended during state extraction at the old host, moving the state from the old to the new host, and installing it at the new host.

Two metrics are used to assess temporal cost: *freeze time* and *latency spikes*. Freeze time quantifies the duration for which an operator cannot work, i.e., freeze time $= t_{start} - t_{stop}$, where t_{stop} is the point in time when the old host stops the operator and t_{start} is when the new host resumes it. Latency spikes quantify the increased latency of event delivery caused by a non-working operator. It is often approximated by the time needed for state movement, which is the duration for which the state is in transit between the old and the new host, i.e., state movement time $= t_{receive} - t_{send}$, where t_{send} is the time at which the old host starts sending the state, and $t_{receive}$ is the time at which the new host has received the entire state. The state movement time depends on the size of the state and the available bandwidth between old and new host. Thus, state size can be seen as related to the costs of both resources and time. Existing research is largely concerned with the tuple delay of a placement [93], but tuple latency caused by migration has not been given the same priority.

It should be noted that latency spikes reflect the cost much better than freeze time since it is possible for the operator not to produce any event during the freeze period. Examples of such a case are incoming events during this period that do not match the pattern that can trigger the operator to produce an event, or if a tumbling window implemented by the operator is much larger than the migration time such that all delayed incoming events can be processed on the new host before the window expires.

From the descriptions of the different types of algorithms above, it is easy to see that they differ in the cost of migration. Operator downtime or the latency of the output tuple can be considered a reasonable definition of the cost of migration for single-track state movement algorithms. However, for parallel-track algorithms, this definition of cost can result in excessively frequent migrations, as it typically results in a value close to zero. Therefore, it is necessary to define the cost of migration in such a way that the migrations do not become too frequent.

With parallel-track, operator replicas need to be executed during migration, and upstream nodes must send duplicate streams to the old and new hosts. They may also take a significant amount of time to execute when using window-recreation [103], which, in addition to using operator replicas during this time, might result in a significant increase in monetary costs. Therefore, it makes sense to consider the monetary cost when using parallel-track algorithms.

C. Migration Decision

To perform the migration decision several steps are necessary (see Figure 6). First, the decision process needs to be triggered. Then, a better placement has to be selected. The cost and benefits of a migration to the host must be estimated and compared in order to determine whether it is worthwhile to migrate.

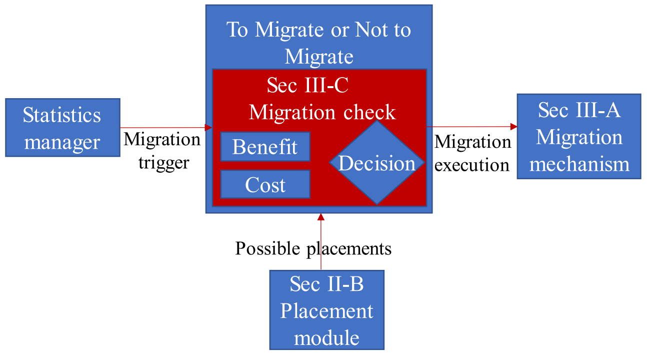

To Migrate or Not to Migrate
Sec III-C Migration check
Benefit
Cost
Decision
Statistics manager
Migration trigger
Migration execution
Sec III-A Migration mechanism
Possible placements
Sec II-B Placement module

Fig. 6. Migration decision-making.

Most studies have handled migration as part of an adaptation mechanism, where the goal is to improve execution or recover it in case of node failure. Regularly collected metrics can be used to indicate the need for an adaptation. Recent surveys have focused on adaptation mechanisms [7], [18], while this tutorial focuses on operator migration. Migration is usually the most costly aspect of an adaptation, and this perspective can be useful for better understanding adaptations. Even though adaptations differ in some aspects, they share major parts in terms of cost.

1) Migration Goal: This section describes four of the most common goals of migration: *load balancing* to distribute the load evenly on the available nodes, *elasticity* to efficiently leverage computational resources, *fault tolerance* to ensure that the DSP system can continue processing in the event of failures, and improving the QoS. This is not a comprehensive set of migration goals but it constitutes the main categories. For instance, security can be another reason for making adaptation changes. However, the general migration goal is to improve performance of the system. For instance, a DDOS attack might cause a migration, but this would also be addressed using QoS as the migration goal.

While all goals of migration can be applied to any deployment environment, the solutions with load balancing and elasticity are mainly aimed at cloud-based DSP systems and executed within a single data center, whereas QoS optimization is normally carried out when an operator undergoes back-pressure and needs an adaptation to improve the QoS, which can happen in any deployment environment. While migration is relevant for fault tolerance, few solutions describe it as a mechanism to facilitate reliable execution. Instead, solutions often use an upstream rollback approach [139] that replays to the new host tuples that are part of the failed operator. Below, we analyze all the goals of migration except fault tolerance.

QoS-driven migration is employed to enhance the system's QoS. Crucial QoS metrics in operator migration include bandwidth, availability, latency, throughput, and more. For instance, to maintain the goal of low latency in mobile settings, the system works to position the operators close to the data producer. This can be seen as a broad optimization problem, with the objective of maximizing the selected QoS metrics, as depicted in Equation (1). Later, in Section V, we delve into which of these metrics are frequently taken into account:

$$\max \sum_{i=1}^{n} QoS_i \tag{1}$$

Another important parameter in mobile settings is energy preservation for resource-constrained nodes [97], [145]. In a cluster setting, the goal is often to ensure that the nodes are not overloaded and that the latency of the tuple is not too high. If the latency of an operator increases significantly, it might be migrated to a node that can provide lower latency.

The basic goal of migration is to improve the QoS. Most solutions are more specific about the goal of migration because finding the optimal solution is usually an NP-hard problem [138], which is unfeasible to solve for networks of most sizes. A simpler approach is to add constraints to the operator. If the operator cannot fulfill these constraints, it must be relocated. This is typically a much more scalable solution that looks for a placement that is good enough, instead of looking for the optimal solution. It is a push-based manner of letting the coordinator know when the operator needs to be relocated. It should be noted that constraints or thresholds are also often used to achieve the other goals of migration. One characteristic of QoS-based migration is that it is mainly related to the migration of individual operators.

Load balancing is a necessity in distributed streaming systems, because the workload might vary significantly over time, leading to unbalanced distributions of state over the processing nodes. The coordinator should monitor the resource usage on the nodes to ensure that neither the network nor the CPU resources become bottlenecks for the performance of the operators. If resource usage on the nodes is unbalanced, the coordinator moves some of the tasks among the nodes. If these are stateful processes, the tasks to be moved must be paused, moved, and restarted on the new node. Load balancing-driven migration differs from QoS-based migration in the sense that the data consumers do not necessarily benefit much from the balancing, and in that multiple operators are usually migrated through load balancing. However, the decision on when to perform load balancing and where to migrate operators must still take into account the same concerns as for constraint violation, i.e., whether the cost of migration is worth the benefit of the new placement.

Load balancing is typically modeled as a resource scheduling problem, where the goal is to distribute the load as evenly as possible. For instance, minmax the latency on the nodes [77], as shown in Equation (2).

$$\min \quad (\max \quad l(G_i)) \tag{2}$$

Elasticity refers to adding or removing operator replicas that facilitate parallel processing (also called operator scaling). For instance, a query with stateful windows that are grouped by a key can be run in parallel in multiple threads, where each thread is responsible for a subset of the keys. Four scaling operations are commonly used:

- *Scale up:* Create a new process and migrate some partitions of existing threads to it.
- *Scale down:* Migrate all partitions of a thread to the other threads and shut it down.

- *Scale out:* Create a new worker to which some threads can be moved.
- *Scale in:* Remove a worker and move its threads to existing workers.

In a cloud setting, the scaling out of a streaming system means adding more servers to a cluster. The streaming system then automatically decides which operators to move to it and potentially scale up. Scaling in means the opposite: A server is removed from the cluster. First, all the server's operators are moved to other servers and some scale-down operations might be performed. Scaling in and out can be modeled as special cases of load balancing. When scaling out, a new container or VM is started on a new machine, which is then added to a load balancing pool. The load balancer can then use this new machine for load balancing. When scaling in, a machine is eliminated from the load balancing pool, and at least its own state must be migrated to the other nodes.

2) Triggers: To determine whether migration should be performed, it is necessary to compare the current placement with an alternative placement to estimate the benefits of migration. If these benefits are significantly greater than the costs, migration is beneficial. However, the calculation of a new placement, its benefits, and the related costs might require a non-negligible amount of resources. As such, the naïve approach to scheduling a migration decision with a fixed frequency might be too costly. Instead, some form of context awareness needs to be supported to detect changes in the system (e.g., related to workload, resource availability, or mobility) that indicate that there might be a good chance of determining a better placement. The relevance of such changes is generally implied by the goal of migration. Monitoring the runtime system is an important task to detect such changes. The DSP system can also perform some book-keeping, like the number of operators a node hosts, and trigger a migration decision if a threshold is reached.

A simple trigger is a constraint or threshold. For instance, load balancing systems may make balancing decisions when the load imbalance of the systems is above a certain threshold. For elasticity-based solutions, checks on whether to scale in or out are similarly performed using thresholds. If the system has a balanced load and its use is still above a given threshold, the system might decide to scale out. If the utilization is below a threshold, the system can scale in. If an operator has latency constraints that are not fulfilled, the coordinator can be notified that migration must occur. The coordinator can either be the node hosting the operator in a decentralized solution or a centralized controller in a data center. In all scenarios, a unit collects metrics from the runtime system in order to make a decision.

3) Timing Decision: Migration decisions can be made reactively or proactively. In the former case, a system migrates when the given situation calls for a change to be made, such as when QoS guarantees for an operator are not fulfilled. Proactive migration decisions rely on predictions about future changes that require migrations.

In several cases, the need for migration scales with its cost. For instance, if the migration is triggered when the tuple rate exceeds a limit and causes QoS violations, more tuples are affected by operator downtime when the need for migration is more pressing. In other words, the more pressing the need to migrate is, the higher the cost of migration is. If a node is over-provisioned, and cannot handle a higher input rate for a given operator, the operator benefits from being migrated to another node. If this situation is detected when the input rate is already too high, a potential migration results in latency spikes for the affected tuples. However, if it is possible to predict that the tuple rate will increase, one can reduce the cost of migration by proactively migrating before the tuple rate becomes too high.

The cost-benefit analysis for making migration decisions is not trivial as the cost of migration is a one-time investment and the benefit from better performance is accumulated over time. When confronted with dynamic surroundings in stream processing scenarios, it makes sense to consider a given placement only for a given amount of time. This time can be regarded as the horizon for which predictions are made. Migration decisions are then made in such a way that the new placement amortizes cost during that time. As such, this time horizon is called amortization time. The notion of working with a limited future horizon for making optimization decisions is also used in model predictive control (MPC), and has been applied by De Matteis and Mencagli [107] to make proactive scaling decisions.

The higher the number of tuples that are impacted, the more the migration option is penalized. However, the number of tuples impacted is an estimate that depends on the accuracy of the prediction. It is possible to assume that tuples are sent evenly across the time window of the horizon, in a single burst, as fast as possible, or a mix between the two. To make such predictions, it is necessary to collect metrics from upstream nodes to determine the density of distribution of the transmitted tuples.

4) Cost Versus Benefit: Once the decision process has been initiated, it is necessary to determine a better placement and relate its benefits to the costs of the migration to determine whether to migrate [13]. One clear approach to calculating a new placement is to re-run the original placement algorithm with the same objective function. Some of the data needed for calculating a new placement might be available from the monitoring component that triggers the migration decision. In most cases, additional live data must be collected, where this represents a substantial part of the overhead of making the migration decision.

The gain in performance owing to a new placement is generally reflected in the output of the objective function of the old placement versus that of the new placement [97], [138], [146]. By optimizing the objective function during placement, a new placement that delivers the best performance is identified. The problem with simply migrating to the host with the best performance is that the cost of migration might be so high that it is not worth migrating. It might be that a sub-optimal placement is preferred in terms of the objective function owing to a lower migration cost, or maybe that no migration is worth it at all. What makes the comparison of cost and placement performance challenging is that they are not directly comparable. On the one hand, multiple, possibly contradicting, metrics can be used to determine cost and performance. On the other hand, the cost of migration is a one-time investment while the performance of a placement represents how a placement performs over a certain amount of time. The placement performance continuously increases the overall benefit as long as there are no changes in the system. As such, there is a need to distinguish between the benefits of placement and migration. The *benefit of placement* simply expresses the difference in placement scores between a new host and the given host, while the *benefit of migration* is calculated based on (1) the cost of migration, (2) the placement performance, and (3) the amortization time.

Three common ways to avoid excessively frequent migrations have been discussed by Lakshmanan et al. [12]: (1) A threshold to ensure that the score of the new placement is significantly better than that of the current placement. (2) If the QoS guarantees of an operator are violated, it triggers a migration, which means that migrations are performed only when necessary. (3) Periodic re-evaluation of the objective function where the interval is set to be reasonably high. In a more recent example, Buddhika et al. [66] regularly calculated interference scores of operators that describe the need for migration, and migrated them to a node where they were subjected to less interference. However, neither Lakshmanan et al. [12] nor Buddhika et al. [66] performed an explicit cost-benefit analysis. This is of interest to us not only to avoid excessively frequent migrations, but to understand why migration is worth it in some cases and not in others based on its costs and benefits. Suppose a placement is an improvement over the given placement. In that case, we want to be able to state exactly why the migration is worth performing (or not) in a meaningful and understandable way. The amortization of the cost of migration is a simple goal to understand as long as one weighs the one-time cost of migration against the benefit of the continuous performance of the new placement, but this deliberation is often not presented explicitly in existing works.

IV. Migration Mechanisms

In this section and Section V, we present an overview of existing literature on operator migration. Specifically, we examine the design of current migration strategies, with a focus on those that assume full consistency of the operator state.

As the volume and velocity of data have increased with the emergence of big data [147], the simple single-track moving state algorithm has become inadequate. Specialized and innovative solutions that provide no downtime and solutions that leverage fault tolerance mechanisms, such as periodically performed back-ups, have been designed. We explore the state-of-the-art migration algorithms and provide a historical perspective on innovations proposed.

Migration mechanisms are characterized by their state and stream management. This involves executing certain tasks, such as redirecting, buffering, pausing streams, and moving states between nodes. Moreover, it is important to specify whether these tasks can be executed in parallel and where it is most beneficial to execute them. The most important properties identified in Section III-A are whether the algorithms require state migration and how this is performed, and whether they are single-track or parallel-track. Most of the investigated migration mechanisms can be derived from these properties. For instance, some mechanisms are centralized, and rely on a coordinator, such as [46], [78], [103], whereas others are decentralized and initiate migration on the operator host, e.g., [85], [96]. In some cases, multiple dependent migrations are planned and performed in sequence, but the details of managing multiple migrations are not presented in this tutorial. Examples of such algorithms include load balancing, where many keys of an operator may be moved to a new location, and when an operator graph is distributed geographically and several operators are migrated, e.g., in TCEP [99].

The most fundamental mechanisms are single-track without state migration, single-track with state migration, and parallel-track without state migration, i.e., window-recreation. These mechanisms were introduced together by Zhu et al. [141] and were later applied to the SPE CAPE [56]. The authors discussed the steps of migration and cost models of the different mechanisms. They called them moving state, parallel-track, and pause-drain-resume migration mechanisms. Using the terminology established in Section III, the moving state mechanism is single-track moving state, the parallel-track mechanism is parallel-track without state migration, and the pause-drain-resume mechanism is single-track without state migration. The paper by Shah et al. [46] forms the basis for load balancing, and presented a means of repartitioning keys in a key-value-partitioned operator state, which is relevant for cluster-based systems. We characterize this mechanism as a single-track moving state algorithm, but in which the operators are already running on the destination node. In contrast to some studies, for instance by Qin et al. [18], we do not consider state movements in load balancing and operator migration to be fundamentally different, and posit that only the entities being migrated are different, i.e., keys are moved instead of operators. In load balancing, the entities being migrated are often a set of keys and their associated states, whereas in operator migration, the entities are usually an operator and its associated state.

A. Mechanism Descriptions

This section describes the relevant mechanisms in a concise and systematic manner. Since details of what happens in migration mechanisms are typically omitted from research papers, our descriptions may deviate to some extent from the original implementations of the migration mechanisms considered. For the most significant variations of these mechanisms, we show how migration is performed using a figure that illustrates the topology of stream processing and the communication between nodes. Please refer to the legend in Figure 7 for the description of symbols that are used in this section to describe the migration mechanisms. The following types of nodes are used: old host (OH), new host (NH), upstream nodes (US), and downstream nodes (DS). The upstream node and downstream node can both represent one

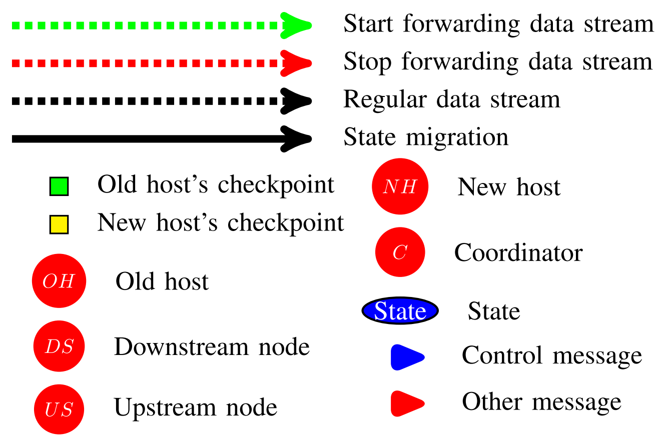


Fig. 7. Legend for the migration mechanism illustrations in Figure 8–12.

or more nodes, but for the sake of simplicity, only one of them is shown in the figures. Each figure is accompanied by an enumerated description of the steps of the relevant algorithm on the right-hand side, and each step is provided in the figure to facilitate the understanding of the algorithm. Furthermore, we show in subsequent listings the contents of the control messages sent to provide the reader with a kind of computational viewpoint of the involved nodes. Since the control messages comprise the tasks that must be executed by the nodes, there is a natural correspondence to the steps listed in the figure.

Control messages used for migration are typically embedded into the data streams. These tell the nodes that a migration will be performed, and might be used for other coordination tasks. In some solutions, this message is sent only to the old or the new host; in other solutions, it is sent to the old and the new host, or to upstream nodes, old and new hosts, and downstream nodes. There may be many reasons for notifying different nodes about migration, such as updating the view of where key partitions are maintained, and routing streams. We describe only a subset of control messages for each mechanism, and they describe the essential tasks that should be executed to perform stream and state management tasks. In the illustrations, the control messages are shown in blue whereas the other messages are shown in red. In addition to the visual representation of the topology and messages sent among nodes, the essential tasks to execute during migration are shown in a listing. The first blue control message from the coordinator, which features in step one of each algorithm, is shown in this listing. All other control messages represent subsequent steps in the algorithm that are described in the first blue control message.

B. Standard Moving State

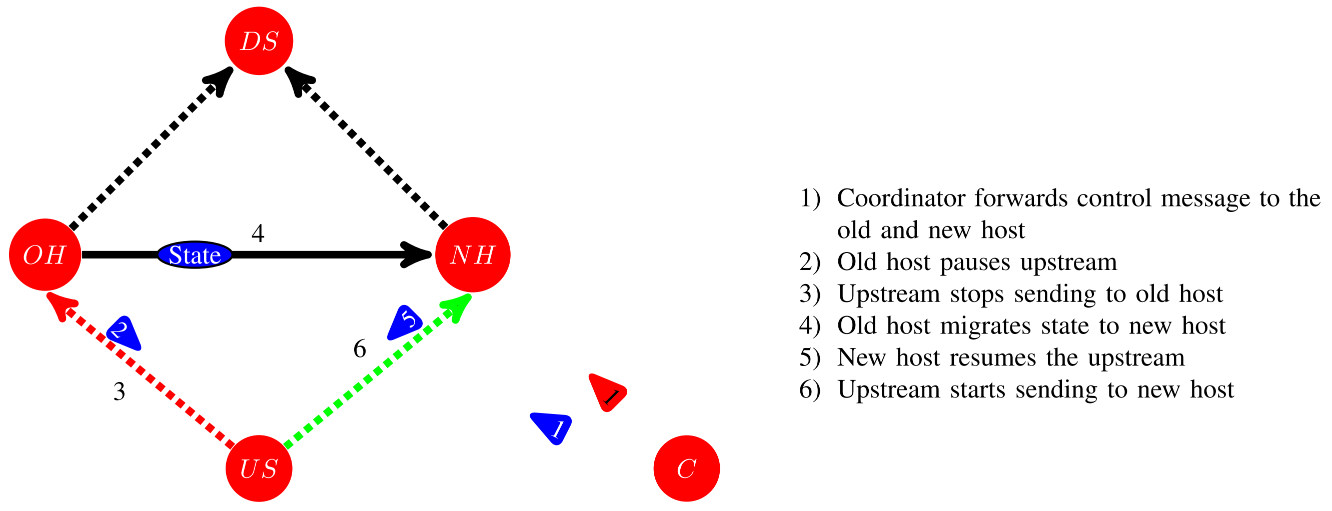
The standard moving state mechanism (see Figure 4 and 5) uses direct state movement between the old and the new hosts, and the entire state is sent all at once. Aside from moving the state, migration requires changing the stream routing. Figure 8 shows the steps involved in the standard moving state algorithm developed by Shah et al. [46]. They proposed an operator called Flux that can adapt the state partitioning of the pipelines of dataflow using a state movement algorithm. Other state movement mechanisms largely follow the same steps, but they might vary in their approach to stream management or in the roles assigned to specific nodes.

Our interpretation of the moving state mechanism's blue migration control message from Step 1 in Figure 8 is described in Listing 1. The upstream nodes buffer, stop, and redirect streams from the old host to the new host. Following this, the task of migrating the state from the old host to the new host is issued to the old host, after which the streams are resumed. Instead of stopping the upstream nodes, other solutions [78], [103] redirect streams from the upstream nodes to the new host. The new host buffers the streams and starts to process them when the state from the old host has been received and installed. Other solutions send the control message to the old host instead of the upstream nodes [85], or even to the new host [50]. The benefit of this class of migration mechanisms is that it is straightforward and simple, but the downside is that it may cause significant downtime.

C. Parallel-Track

There are two types of parallel-track mechanisms: state-recreation and window-recreation mechanisms. The difference between them is that state-recreation involves moving state and window-recreation does not. The completion time for a state-recreation algorithm is proportional to the state size, whereas for a window-recreation algorithm, it is proportional to the window size [103]. Zhu et al. [141] introduced the window-recreation parallel-track migration mechanism. Gulisano et al. [103] presented both a window-recreation and a state-recreation mechanism, and Ottenwälder et al. [96] performed state-recreation migrations based on changes in mobility. Madsen et al. proposed a direct window-recreation mechanism in Enorm [62] and a checkpoint-assisted state-recreation mechanism in [58]. ChronoStream [113] performs a checkpoint-assisted state-recreation migration of state slices to provide horizontal elasticity. UniMiCo [144] (uninterruptable migration of continuous queries) is a direct window-recreation algorithm that can handle both time-based and tuple-based window semantics.

StreamCloud's [103] state-recreation and window-recreation mechanisms are shown in Figure 9 and Figure 10. In both mechanisms, a handover between the old and new hosts is scheduled using a timestamp. In window-recreation, the handover is performed in a way such that the old host empties its windows and the new host fills them in parallel, resulting in a smooth handover. For this purpose, the upstream nodes send tuples to both the old and new hosts. In state-recreation, the old host sets the handover timestamp immediately before serializing and transmitting the state to the new host. Any subsequent tuples with a timestamp lower than the handover timestamp are processed by the old host, and the other tuples are processed by the new host. Operator downtime can be avoided here if the handover timestamp is set to a time after the new host is expected to have received the state and started its execution. When the state is received by the new host, it processes all tuples it receives from the upstream nodes in



1) Coordinator forwards control message to the old and new host
2) Old host pauses upstream
3) Upstream stops sending to old host
4) Old host migrates state to new host
5) New host resumes the upstream
6) Upstream starts sending to new host

Fig. 8. Moving state (according to [46]).

```
ControlMessage(OH
   ControlMessage(Upstream
      BufferStreams(Streams(query))
      StopStreams(Streams(query))
      Redirect(Streams(query), OH, NH)
      ControlMessage(OH, MoveState(query, NH))
      Resume(Streams(query))))
```

Listing 1. Single-track moving state.

parallel with the old host, but produces only tuples caused by input tuples with a timestamp higher than the handover timestamp.

Our interpretation of the window-recreation mechanism for the blue migration control message from Step 1 in Figure 9 is described in Listing 3. The control message is sent by the coordinator to the upstream nodes. From there, it is forwarded to the old host, which schedules the takeover time for the new host and sends it to the upstream nodes. From then on, the upstream nodes send tuples to both the old and the new hosts. The new host processes the same tuples as the old host, but does not produce any tuple until the old host has stopped processing.

Our interpretation of the state-recreation mechanism for the blue migration control message from Step 1 in Figure 10 is described in Listing 3. The control message is sent by the coordinator to the upstream nodes. This algorithm requires slightly greater coordination between the old and the new host than in case of window-recreation, because the old host must move its state to the new host, and the latter needs to know the takeover time. These classes of migration mechanisms have as benefit that they may result in zero downtime for the operators, but at the expense of overhead when duplicating the data streams and maintaining two copies of the stream processing system. Window-recreation requires no state transfer, but at the expense of increasing the total migration time, which in a pay-as-you-go scenario increases the monetary cost of running the system.

D. Indirect State Movement

Gedik et al. [142] described an indirect state migration mechanism for load balancing that has been used as the basis in several studies [61], [63], [107]. They proposed an operator that outputs to multiple replicas partitioned by keys, called a splitter, that can decide to change the distribution of the keys, which requires state migration between replicas. Moreover, they introduced a two-phase approach to migration: donate and collect. In the donate phase, the state to be migrated is moved from the old host's in-memory store to a backing store. In the collect phase, the new host retrieves the state from the backing store. This method was subsequently used by Cardellini et al. [61] and Li et al. [63] to implement features of elasticity in migration in Apache Storm. The drawback of this method is that streams from the upstream nodes are paused during execution. De Matteis and Mencagli [105], [107] defined a similar state migration mechanism. However, their implementation contains a number of improvements, e.g., the splitter can send new tuples during state movement instead of blocking until migration is complete.

In the donate phase of the mechanism proposed by Gedik et al. [142], replicas place the state to be moved into packages, one for each replica that takes over the state. The data are moved away from the in-memory store of the replicas to a backing store. A vertical barrier is used across the replicas to ensure that they do not progress to the next phase until all packages have been donated. In the collect phase, the replicas check the backing store for any packages that contain the state that they take over and restore it. Following this, a horizontal barrier is used to prevent the splitter from sending any tuples until the migration process has been completed.

The benefit of the two-phase approach is that it involves an API where an operator simply requires implementing methods to extract the state, and sends it to a backing store instead of requiring intricate communication among operators. Moreover, it can use existing fault tolerance mechanisms that periodically create checkpoints of states for the backing store.

E. Partial State Movement

With partial state movement, the state is partitioned and each partition is moved individually, to minimize operator downtime. MigCEP [96] is an algorithm designed for frequent migrations to minimize downtime. The state is split up into

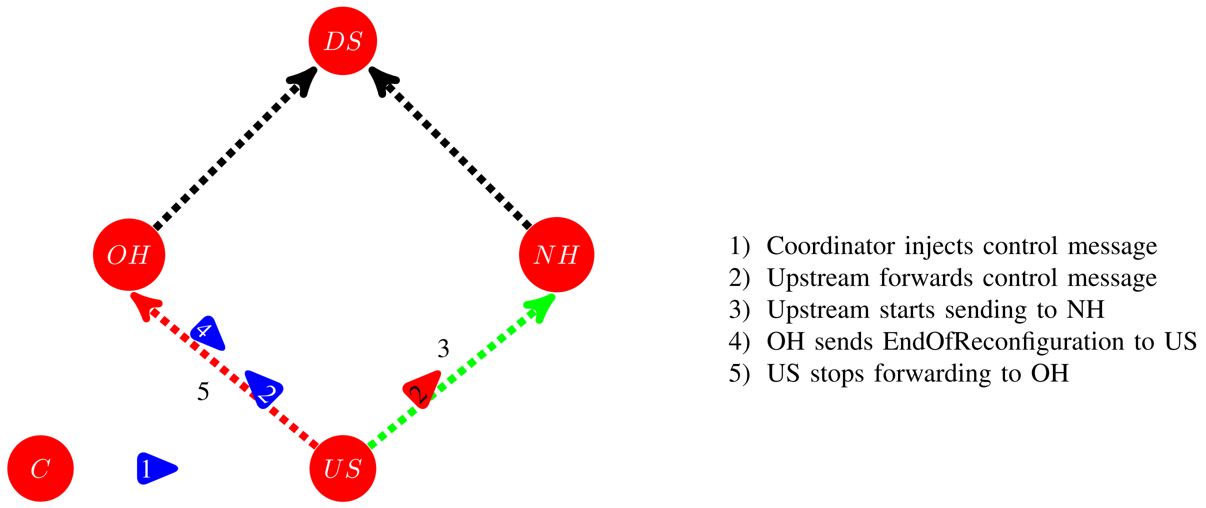

DS
OH
NH
C
US
1
2
2
3
4
5
1) Coordinator injects control message
2) Upstream forwards control message
3) Upstream starts sending to NH
4) OH sends EndOfReconfiguration to US
5) US stops forwarding to OH

Fig. 9. Parallel-track window-recreation algorithm (according to [103]).

```
ControlMessage(Upstream
  ControlMessage(NH,
    StartQuery(query))
  ControlMessage(OH
    ControlMessage(Upstream,
      Schedule(RemoveNextHop(Streams(query), OH)
                TakeoverTime(query))
      AddNextHop(Streams(Upstream), NH))))
```

Listing 2. Window-recreation.

two parts: immutable and mutable. An immutable or static state includes the operator and, possibly, databases whose data have not changed during migration. A mutable state consists of tuples that are being processed in the operator.

A further improvement involves sending the last incremental checkpoint of the state to the new host before the operator goes down. This is the case in ChronoStream [113] and Rhino [78], where the state is split before the operator is migrated, and an incremental checkpoint that includes the new state after the first part has been extracted. This can be seen as analogous to the immutable and mutable states described in MigCEP [96].

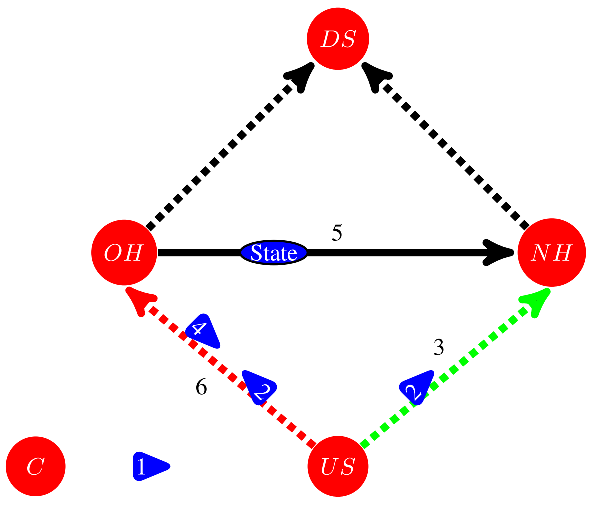
Megaphone [75] is a state migration technique for migrating many keys in an efficient way to minimize latency spikes. In this case, the state is split into many equal-sized parts. Each causes some downtime for the system. However, while the total migration time increases, the spikes due to tuple latency are substantially reduced compared to sending the entire state all at once. However, the Megaphone mechanism introduces some additional overhead to operators during non-rescaling periods. Another state migration technique called Meces [80] may improve upon Megaphone by being more lightweight and prioritizing the migration of partial states that are needed by incoming tuples. A newly received tuple that requires a given partial state fetches it from the old host, instead of waiting for it in a larger batch, or relying on complex synchronization mechanisms. The commonality between Megaphone and Meces is that they are only beneficial with operators where the state is split up into many keys, e.g., word count with words as keys, equijoin operators, or other aggregation operators with keys.

Fragkoulis et al. [7] distinguished between all-at-once and continuous state movements, which are classified in this tutorial as all-at-once and partial state movements, respectively. Megaphone, Rhino, and ChronoStream are characterized in this tutorial as exemplars of partial state movement, while Fragkoulis et al. categorized Megaphone as using continuous state movement, and Rhino and ChronoStream as using all-at-once state movement. The reason for this difference is that ChronoStream and Rhino rely on *distributed checkpoint replication*, and need only to send the state that has been built up since the last checkpoint. In this tutorial, migration is further divided by distinguishing between solutions that use distributed checkpoint replication and those that do not. Megaphone and Meces do not use it, and send the entire state directly from the old host to the new host, whereas ChronoStream and Rhino depend on distributed checkpoint replication. If Rhino and ChronoStream do not use distributed checkpoint replication, this means that the initial checkpoint is sent from the old host to the new host instead of existing on the new host already. Therefore, using partial state movement is not necessarily an indication that multiple states are sent during migration.

State shedding combines load shedding and operator migration in a way that demands fine-grained migration [143]. Each partial state is assigned a utility based on how important it is, similar to how it is done in load shedding. The most important partial states are then migrated, whereas the least useful partial states may be dropped. The primary advantage of state shedding is the ability to prioritize the migration of critical states. However, it does require calculating the utility of partial states, which can be challenging to predict.

F. Distributed Checkpoint Replication

Some solutions leverage fault tolerance mechanisms to improve the scalability and performance of migration using periodically updated, and distributed and replicated checkpoints of the state of stream processing. Since these algorithms use checkpoint solutions that may already exist, they are called



1) Coordinator injects control message
2) Upstream forwards control message
3) Upstream starts sending to NH
4) OH sends EndOfReconfiguration to US
5) OH migrates state to NH
6) Upstream stops sending to OH

Fig. 10. Parallel-track state-recreation algorithm (according to [103]).

```
ControlMessage(Upstream
  ControlMessage(OH
    ControlMessage(NH,
      StopStreams(OutputStreams(query))
      StartQuery(query)
      Schedule(TakeoverTime(query)
               StartStreams(Streams(query))))
    ControlMessage(Upstream,
      Schedule(RemoveNextHop(Streams(query), OH),
               TakeoverTime(query))
      AddNextHop(Streams(Upstream), NH)))
  MoveState(query, NH))
```

Listing 3. State-recreation.

checkpoint-assisted algorithms. If the target of migration is a host that already contains the state, a migration algorithm can be as simple as one that loads the checkpoint in memory and replays the upstream tuples to the new host. This requires exactly-once guarantees, as provided by pub-sub systems such as Kafka [140]. A parallel-track algorithm can work similarly, but, instead of stopping the old host before replaying tuples on the new host, both the old host and the new host run until the latter takes over. In this process, output tuples need to be filtered to remove duplicate tuples. ChronoStream [113] uses distributed checkpoint replication to implement a parallel-track algorithm, Rhino [78] to realize a single-track algorithm, and the proposal of Madsen and Zhou [58] to carry out both. The algorithms often also use partial state movement when updating checkpoint replicas to send as little state as possible.

Monte et al. [78] introduced a checkpoint-assisted single-track migration mechanism called Rhino that can migrate state sizes of up to terabytes 15 times faster than the state-of-the-art solutions (as of 2020) by using incremental checkpointing. Their algorithm is shown in Figure 11. Most of the state is sent before the old host is stopped. Afterward, it sends an incremental checkpoint that represents a change in the original state. In this way, only tuples that arrive after migration has started need to be migrated in the incremental checkpoint. This algorithm is a cluster-based migration mechanism that is executed by a handover manager (HM). The HM informs all workers about the migration and about what will happen, by injecting a control message into the source streams (inspired by Chi [71]). Afterward, the source nodes are redirected. When the old host has received a control message on all of its incoming streams, it sends the state to the new host. The green box indicates the state repository for the old host, while the yellow box indicates the state repository for the new host. The intermediary hosts, including the old and the new host, send control messages to their next hop nodes. When the nodes have completed their tasks, including the redirection of streams and the migration of state, they acknowledge the HM. The migration is complete when all nodes have acknowledged the HM.

Our interpretation of the checkpoint-assisted moving state algorithm's blue migration control message from Step 1 in Figure 11 is described in Listing 4. The control message is issued to the upstream nodes, which forward a control message to all downstream nodes. We describe tasks that the old host might be assigned. The main difference between this algorithm and the standard moving state algorithm is that most of the state is assumed to be on the new host before the migration starts. As such, when the state is moved, it is moved using the partial state movement task `MoveIncrementalState` instead of `MoveState`.

Wu et al. proposed ChronoStream [113], a checkpoint-assisted state-recreation migration algorithm that provides horizontal elasticity, as illustrated in Figure 12. The states of all tasks on a node are periodically backed up and sent to the other nodes. As a result, migration only involves updating a subset of the backed-up state, which significantly reduces the number of states to be moved. This process is split into four phases: migration preparation, state rebuilding, dataflow rerouting, and resource release. The first phase sets up a container for the operator on the destination node if this has not been done already. In the second phase, the new host fetches the operator's state locally or remotely and rebuilds it, and notifies the master node when finished. The green box indicates the state repository for the old host, while the yellow box indicates the state repository for the new host. The third phase involves the master telling the data sources to send

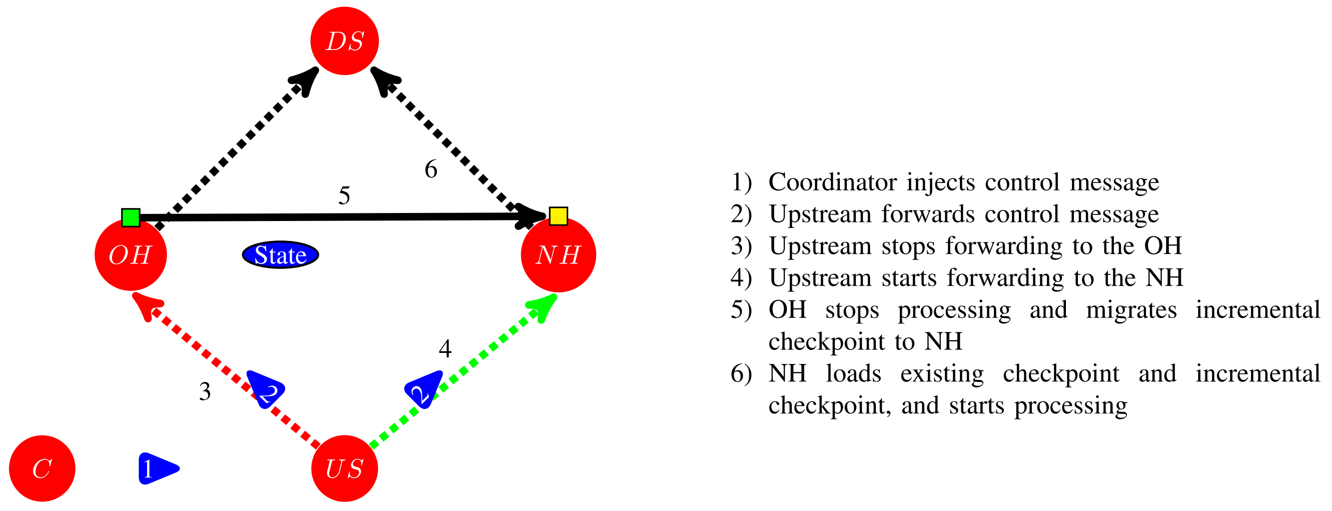

DS
6
5
OH
State
NH
4
3
2
2
C
1
US
1) Coordinator injects control message
2) Upstream forwards control message
3) Upstream stops forwarding to the OH
4) Upstream starts forwarding to the NH
5) OH stops processing and migrates incremental checkpoint to NH
6) NH loads existing checkpoint and incremental checkpoint, and starts processing

Fig. 11. Checkpoint-assisted single-track mechanism (according to [78]).

```
# Bootstrapping
ControlMessage(OH, ReplicateCheckpoint(NH))

# Migration
ControlMessage(Upstream
  ControlMessage(NH,
    BufferStreams(NH, Streams(query))
    StopStreams(NH, Streams(query)))
  ControlMessage(OH,
    Redirect(Streams(query), OH, NH)
    MoveIncrementalState(query, NH)
    ControlMessage(NH, StartStreams(Streams(query)))
))
```

Listing 4. Checkpoint-assisted single-track.

tuples to the new host as well, including any tuple that is not included in the state that the new host received. At this point, the new host participates in the processing and produces the same tuples as the old host, and duplicate output tuples are filtered out by downstream operators based on the sequence numbers of the tuples. Finally, the controller tells the old host to release the resources such that the new host is the only node running the operator.

Our interpretation of the checkpoint-assisted parallel-track mechanism for the blue migration control message from Step 1 in Figure 12 is described in Listing 4. The main difference between the parallel-track checkpoint-assisted mechanism and a non-checkpoint-assisted mechanism is that the immutable state is sent or made available on the new host before any downtime occurs.

Checkpoint-assisted migration mechanisms can greatly reduce operator downtime, providing a significant performance boost in cases where fault tolerance features are already established. However, if the system does not already have checkpointing functionality, it must be added.

V. Migration Decision

We review the elements of migration decision-making including the calculation of the costs and benefits of migration. There are existing surveys that go into depth of what methods are used for decision-making [19], and that is not the purpose of this tutorial. We introduce some migration decision methods and describe which metrics are used to base migration decisions on, and what measurements are done in evaluations. This gives a picture of the struggle of making decisions on incomplete data, because the consequences of a migration choice are not known until it is done.

A. How to Make Migration Decisions

To start out, we give a small introduction to the topic of how to make migration decisions. We can split this problem in two phases: the problem definition phase and the problem solution phase. The problem definition defines mathematically what is the goal of the system, e.g., to minimize load imbalance, latency and maximize throughput. Thereafter, the problem solution involves some way to achieve this. Re-placement or re-scheduling of operators on different nodes or CPU cores is an NP-hard problem. Some works [92], [138] solve the problem using an Integer Linear Programming (ILP) solver such as gurobi [148] or CPLEX [149]. However, they do not scale well since they require a global view of the network. Operator scaling, on the other hand, is a problem about minimizing the amount of instances to operators while upholding the QoS guarantees [150].

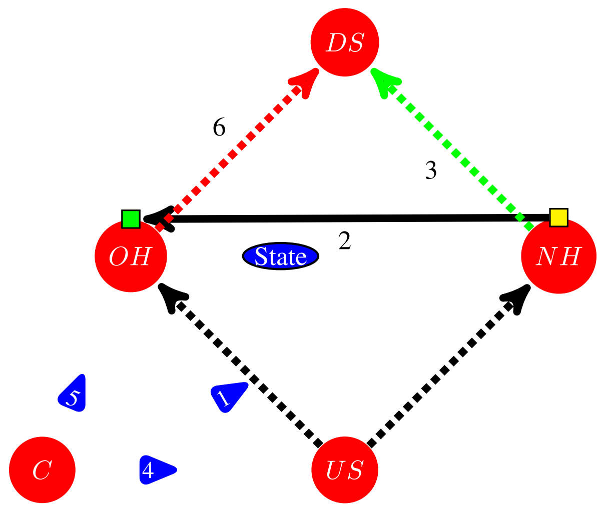
In Table IX, we summarize a selection of the studied papers that contribute to the modeling techniques, algorithms, and decision-making strategies for operator migration. For each paper, we highlight the specific problem being addressed, the applied algorithm or technique, and the approach to decision-making.

The proposed solutions typically use heuristics, which are approximate methods that are designed to quickly find a solution that is close to optimal. These heuristics often involve using specific algorithms that are tailored to the particular optimization problem at hand. For example, a common objective in migration is to minimize the latency of data tuples or maximize the rate at which data tuples are processed.

Pietzuch et al. [117] define a stream-based overlay network (SBON) that uses a placement and adaptation algorithm called

TABLE IX
MODELING TECHNIQUES AND ALGORITHMS FOR PERFORMING DECISION-MAKING IN OPERATOR MIGRATION

Paper Reference	**Problem**	**Algorithm/Technique**	**Decision-Making**
Cardellini et al. [138]	Re-placement or re-scheduling of operators	ILP solver (Gurobi)	Minimize load imbalance, latency; Maximize throughput
Jonathan et al. [92]	Re-placement or re-scheduling of operators	ILP solver (CPLEX)	Minimize load imbalance, latency; Maximize throughput
Pietzuch et al. [117]	Optimization of network usage	Relaxation (Heuristic)	Minimize the network usage of a query
Buddhika et al. [66]	Load balancing	Prediction rings (Heuristic)	Reduce interference that negatively impacts performance
Gedik et al. [57]	Load balancing	Scan, redist, readj (Heuristics)	Balance the mapping from key to server
Hochreiner [65]	Resource scaling	Threshold-based heuristic	Scale resources up or down based on CPU utilization



1) Coordinator tells NH to upgrade slice
2) NH fetches state
3) New host start processing
4) Controller tells upstream nodes to re-forward tuples
5) Controller tells OH to release resources
6) Old host stops processing

Fig. 12. Checkpoint-assisted parallel-track algorithm (according to [113]).

```
# Bootstrapping
ControlMessage(OH, ReplicateCheckpoint(NH))

# Migration
ControlMessage(US, AddNextHop(Streams(query), NH)
  RemoveNextHop(Streams(query), OH))
ControlMessage(NH,
  ControlMessage(OH, MoveImmutableState(query, NH)))
ControlMessage(OH, StopQuery(query))
```

Listing 5. Checkpoint-assisted parallel-track.

Relaxation. Relaxation is a heuristic algorithm that minimizes the network usage of a query. The overlay network consists of a cost space with three latency dimensions (each direction), and one load dimension. The latency dimensions constitute the latency space, and physical nodes are placed in this space such that the distance between nodes represents the communication latency.

Buddhika et al. [66] present a heuristic algorithm for load balancing where the goal is to reduce interference that negatively impacts the performance of stream processing performance. A construct called prediction rings is applied that predicts the future resource usage of stream processing computations, and these are used to calculate the interference score. Thereafter, the goal is to move stream processing computations to the nodes with the least interference. If the interference score exceeds a predefined threshold, the operator is migrated to a node with less interference.

Gedik et al. [57] introduced three heuristic partitioning algorithms to perform load balancing, namely scan, redist and readj. The job of the partitioning functions is to make sure that the mapping from key to server is balanced. These have in common that they apply the same metrics for making the decisions and have the same end-goal, but have different ways of achieving it.

Hochreiner et al. [65] proposed a platform for elastic stream processing, called PESP, which uses heuristics with predefined thresholds to make scaling decisions. Specifically, when the CPU utilization of the system exceeds a certain threshold, PESP scales up the resources to handle the increased workload. Conversely, when the CPU utilization drops below a certain threshold, PESP scales down the resources to avoid overprovisioning.

B. Parameters

We first provide an overview of the parameters of optimization, and the cost and benefit metrics used in existing work (see Table X and the pie charts in Figure 13. We then describe (1) how cost values are modeled and measured, (2) approaches for optimization to increase benefits, and (3) reactive and proactive methods. The figure and table show similar results to Figure 8 by Cardellini et al. in [21], but here we go more in depth of how the metrics are used.

Even though there are many different definitions of the parameters of optimization, they are often related. Therefore,

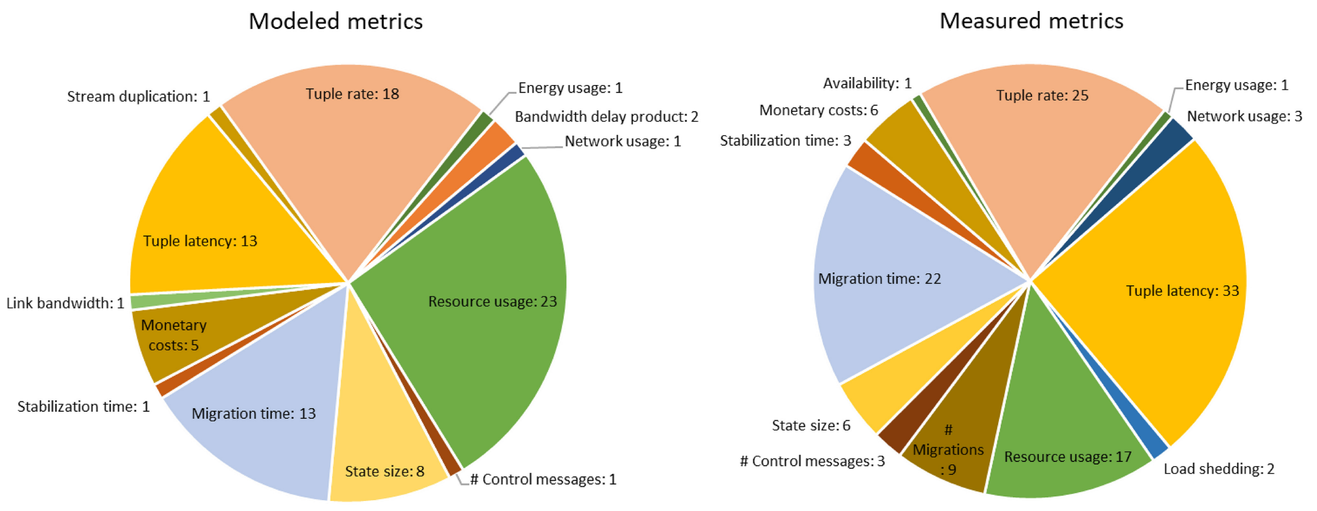


Fig. 13. Popularity of metrics for modeling and measuring migration cost and placement benefit, shown by usage frequency.

TABLE X
GOALS OF OPTIMIZATION GROUPED BY THE GOAL OF MIGRATION

Parameter	Migration goal	Papers
Tuple performance	Load balancing	[50, 64, 66, 87, 104, 104]
	Elasticity	[50, 55, 79, 81, 82, 91, 101, 104, 107, 111, 114, 115]
	QoS	[92, 104]
Network performance	Load balancing	[52, 87]
	Elasticity	[60]
	QoS	[49, 72, 87, 88, 90, 95, 96, 98, 117]
Load	Load balancing	[47, 50, 52, 57, 64, 66, 73, 76, 77, 86, 87, 93, 104]
	Elasticity	[50, 60, 65, 68, 89, 104, 107, 111]
	QoS	[49, 68, 86, 90, 95, 104, 117]
	Fault tolerance	[48]
Migration costs	Load balancing	[52, 57, 73, 76, 86]
	Elasticity	[60, 89, 107, 111]
	QoS	[86, 95, 118]
Monetary costs	Elasticity	[60, 65, 82, 91]
	QoS	[72, 90]
Energy usage	Load balancing	[102]

we group them in Table X into six categories: network performance (e.g., bandwidth, bandwidth latency product), tuple performance (e.g., tuple latency, tuple rate), load, costs of migration, monetary costs, and energy usage. Since the goal of migration is important for optimization, we differentiate between the categories of parameters of optimization in research according to the goals of migration. The most prominent goal is load balancing, and load is the most commonly used optimization parameter. While monetary cost is not commonly used as optimization parameter, migration is often used to avoid the need for over-provisioning and, thus, indirectly reduces monetary costs.

Figure 13 and Table XI give an overview of the metrics used to define the modeled and measured costs of migration, and the modeled and measured benefit of migration. Table XI catalogs each paper by the specific environment for migration, the goal of migration, the cost models, the benefits, and actual costs and benefits measured. This table offers a quick reference to understand how various researchers have modeled and evaluated their solutions. When a metric is used for modeling the migration cost or placement benefit, it means that it is part of an equation that is typically applied to decision-making. This may include attempts at calculating the current system state, or predicting future system state doing proactive migration decisions. Ideally, the measured and modeled metrics should be identical, but they are not. One reason for this mismatch is, that it is much easier to measure values for certain metrics, than to use them for decision-making, like tuple latency and tuple rate. Kalavri et al. [150] discuss how the observed tuple rates may not be good for doing decision-making, because what is really interesting is to know the capacity of a system,5 or the "true" tuple rate. Values for costs and benefits need to be estimated for each migration decision, whereas values of the evaluation metrics are measured during migration. The mismatch between the modeled cost and the benefit, and the measured evaluation metrics might also help to complement future migration decisions and the assessment of migration using further metrics. The most commonly used parameters to determine the cost of migration are the migration time and state size, and few systems use more precise cost parameters, such as latency spike and performance penalty. While some approaches, such as [52], are listed in Table X for performing placement optimization based on the cost of migration, they are notably absent from Table XI because these approaches do not use any specific metric to describe the cost of migration.

TABLE XI
OVERVIEW OF PAPERS ON MIGRATION DECISIONS, COVERING DEPLOYMENT ENVIRONMENT, MIGRATION GOALS, AND METRICS USED FOR MIGRATION COST AND BENEFIT

Paper	**Deployment environment**	**Migration goal**	**Modeled migration cost**	**Measured migration cost**	**Modeled placement benefit**	**Measured benefit of migration**
[94]	Edge	Fault tolerance		Migration time, Stabilization time		
[66]	Cloud	Load balancing		# Migrations	Tuple latency, Tuple rate, Resource usage	Tuple latency, Tuple rate, Resource usage
[61]	Cloud	Elasticity		State size, Migration time		Tuple latency, Resource usage
[138]						Tuple latency, Tuple rate
[89]	Fog	Elasticity		Tuple latency	Resource usage	Tuple latency, Tuple rate
[72]	Cloud	QoS	Migration time	Tuple latency, Migration time, Monetary costs	Tuple latency, Monetary costs	Monetary costs
[107]		Load balancing, Elasticity, QoS		Tuple latency, Resource usage, # Migrations	Tuple rate	
[109]		Load balancing			Tuple rate	Tuple rate
[78]	Cloud	Load balancing, Elasticity, Fault tolerance, QoS		Tuple latency, Tuple rate, Resource usage		Tuple latency, Resource usage
[70]	Cloud	Load balancing	State size			Tuple rate
[73]	Cloud	Load balancing	State size	State size	Resource usage	Tuple latency, Tuple rate
[51]	Cloud			Migration time		
[142]						Tuple rate, Resource usage
[57]	Cloud	Load balancing	State size	State size	Resource usage	
[81]	Cloud	Elasticity	Migration time, Stabilization time	Migration time	Tuple latency, Tuple rate	Tuple latency, Tuple rate, Resource usage
[80]	Cloud	Elasticity		Network usage, Resource usage, Migration time		Tuple latency, Tuple rate
[103]		Load balancing, Elasticity		State size, Migration time		
[111]		Load balancing, Elasticity		Migration time		Tuple latency, Tuple rate, Resource usage
[54]	Cloud			Tuple latency		
[55]	Cloud	Elasticity	Tuple latency	Tuple latency, Resource usage	Tuple rate	
[151]					Tuple rate	Tuple rate, Resource usage
[90]	Fog	QoS	Monetary costs	Monetary costs	Resource usage, Monetary costs	Tuple latency, Resource usage, Availability, Monetary costs
[65]	Cloud	Elasticity			Resource usage, Monetary costs	Monetary costs
[75]	Cloud			Resource usage, Migration time		Tuple latency, Resource usage
[50]	Cloud	Load balancing, Elasticity	State size	Migration time	Stream duplication, Tuple rate, Resource usage	
[48]	Cloud	Fault tolerance		Migration time	Resource usage	
[92]	Fog	Elasticity, QoS	Migration time	Migration time, Stabilization time	Link bandwidth, Tuple latency, Tuple rate	Tuple latency, Load shedding
[95]	Edge	QoS			Resource usage, Energy usage	
[52]	Cloud	Load balancing			Resource usage	
[110]		Load balancing, Elasticity				Tuple latency, Tuple rate, Resource usage
[64]	Cloud	Load balancing	State size		Tuple latency, Tuple rate, Resource usage	Tuple latency
[74]	Cloud	QoS				Tuple latency, Tuple rate, Resource usage, Load shedding
[101]	Edge	Elasticity, Fault tolerance			Tuple rate	Tuple latency

(Continued)

TABLE XI
(*Continued*) OVERVIEW OF PAPERS ON MIGRATION DECISIONS, COVERING DEPLOYMENT ENVIRONMENT, MIGRATION GOALS, AND METRICS USED FOR MIGRATION COST AND BENEFIT

Paper	**Deployment environment**	**Migration goal**	**Modeled migration cost**	**Measured migration cost**	**Modeled placement benefit**	**Measured benefit of migration**
[118]		QoS		# Migrations		Tuple rate
[82]	Cloud			# Migrations	Tuple rate	Monetary costs
[104]		Load balancing, Elasticity, QoS			Tuple latency, Resource usage	Tuple rate
[68]	Cloud	Elasticity, QoS	Migration time	Tuple rate	Tuple rate, Resource usage	Tuple latency, Tuple rate, Resource usage
[99]	Edge	QoS	# Control messages, Migration time	State size, Migration time		Tuple latency
[108]		Load balancing	Resource usage, Migration time			
[58]	Cloud	Load balancing	Migration time	Tuple latency, Migration time		
[62]	Cloud	Load balancing	Migration time	Tuple latency, Tuple rate, Migration time		
[67]	Cloud	Elasticity, QoS	State size			
[59]	Cloud	Fault tolerance	Migration time	Migration time		
[115]		Elasticity				Tuple rate
[102]	Edge	Load balancing				Energy usage
[96]	Edge	QoS	Bandwidth delay product			
[98]	Edge	QoS	Bandwidth delay product			
[117]		QoS		# Migrations	Resource usage	Network usage, Tuple latency
[85]	Fog	Load balancing		# Migrations, # Control messages		
[86]	Fog	Load balancing, QoS		# Migrations	Tuple latency, Tuple rate, Resource usage	Tuple latency, Tuple rate
[88]	Fog	QoS		# Migrations, # Control messages		
[91]	Fog	Elasticity			Tuple latency, Monetary costs	Monetary costs
[56]	Cloud			Migration time		
[46]	Cloud	Load balancing		Tuple latency, Tuple rate		Tuple latency, Tuple rate
[77]	Cloud	Load balancing			Tuple latency, Tuple rate, Resource usage	Tuple latency
[116]		Elasticity			Tuple latency, Tuple rate, Migration time	
[106]		Load balancing, QoS		# Migrations, # Control messages		
[87]	Fog	Load balancing, QoS			Network usage, Resource usage	Network usage, Resource usage
[69]	Cloud	Load balancing	State size	Migration time		
[76]	Cloud	Load balancing, Fault tolerance	State size	Tuple latency, Tuple rate, Migration time	Resource usage	Tuple latency, Tuple rate
[113]		Elasticity, Fault tolerance		Migration time		Tuple latency, Tuple rate
[47]	Cloud	Load balancing	Migration time		Resource usage	
[114]		Elasticity		Migration time, Stabilization time	Tuple rate	
[100]	Edge				Tuple latency, Tuple rate, Resource usage	Tuple latency, Tuple rate
[60]	Cloud	Load balancing, Elasticity	Migration time, Monetary costs	Migration time	Tuple latency, Resource usage	Monetary costs
[79]	Cloud				Tuple rate	Tuple latency, Tuple rate, Resource usage
[93]	Edge	Load balancing		State size		Tuple latency
[49]	Cloud	QoS			Resource usage	
[141]			Migration time			

C. Migration Cost

Accurately defining the cost of migration is essential for making the correct migration decisions. It can be manifested as increased resource consumption and any kind of degradation of execution, such as decreased throughput or increased tuple latency. Table XI indicates that it is more common to measure the cost of migration than it is to model it for making migration decisions.

The vast majority of solutions use migration-specific metrics to model the cost, as opposed to metrics that are used for measuring the benefit of the adaptation. Tuple processing performance is used in many cases to model and measure benefit, but very few approaches have used it to calculate the cost of migration. Heinze et al. [55] modeled and predicted tuple latency as part of the cost of migration, by using the predicted input rate, migration time, and time before queued events can be processed. The tuple latency is defined in Equation (3).

$$latSpike(op) = pause_{Time}(op) + delay_{proc}(op) - delay_{arrival}(op) \quad (3)$$

No solution was provided to model the tuple rate, because it is much easier to measure tuple processing performance than to predict it, when it comes to the cost of migration. Operator downtime is an indicator of spikes in latency but also depends on the tuple rate, because only those tuples that are supposed to be processed during operator downtime are affected. As we discuss later, several solutions have been proposed to model and predict the future tuple rate in a DSP. These predictions can be leveraged to make migration decisions. However, none of the existing solutions take into account the cost of migration in terms of reduced tuple rate.

The migration time and the size of the state to be moved are the most common costs of migration metrics. Migration time is typically calculated as a function of state size, bandwidth, and latency. In environments where the bandwidth and latency are stable, such as within data centers, the state size is often interchangeable with migration time. In most cases, the migration time is assumed to be easy to model and no calculation for it is given. Some solutions can migrate multiple operators at a time, and thus define the migration time as the maximum time it takes to move any of the operators [72], [92]. Cardellini et al. [72] used a data center-based solution to define the operator downtime based on the type of adaptation made, size of the state to be moved, and the round-trip delay between the nodes and the computational resources. WASP [92] is a wide area network solution that defines the time it takes to move an operator based on the state size and bandwidth between links, the latter of which is significantly more limited and variable in a wide area network than a data center. Using the migration time, WASP [92] makes the decision of where to migrate by solving a *minmax* problem by minimizing the slowest migration: $minmax(\frac{|state_{s1}|}{B_{\mathrm{s1}}^{\mathrm{s2}}})$, where $B_{\mathrm{s1}}^{\mathrm{s2}}$ represents the available bandwidth from site s1 to s2 and $|state_{s1}|$ represents the size of the state on the old host (s1).

Zhu et al. [141] focused on the time needed for each step of migration, such as the time spent cleaning the accumulated tuples, state matching, moving the state, and recomputing it.

In Elysium [68], migration time is defined as: $R_{state} + R_{restart} + R_{queue}$, where R_{state} is the time it takes to send the state, $R_{restart}$ is the time it takes to restart the topologies, and R_{queue} is the time it takes to process the tuples that were received and queued up during $R_{state} + R_{restart}$.

Ma et al. [108] defined a migration cost model: $cost = (t2 - t1) \times (i2 - i1)$, where $t2 - t1$ is the migration time, and $i2 - i1$ is the difference in performance between the new host and the old host. The logic is that migration decisions need to make a trade off between the migration cost and benefit. If the new host has significantly better performance than the old host, it might be worth to do the migration, even though the migration takes a long time.

Using state size as the cost of migration is among the easiest ways of defining cost because it requires only looking at the size of the state to be migrated. The solutions that we analyzed that use state size as cost metric are all cloud-based, which makes sense since data centers feature a high and stable bandwidth between nodes, in contrast to geo-distributed environments. The state size is frequently used as part of the objective function when making migration decisions [50], [57] as part of a constraint to prevent costly solutions from being selected [67], and can even be the only criterion to minimize when making load balancing decisions [70], [73].

Luthra et al. [99] used the number of control messages during migration as part of the definition of the cost of migration. This parameter is significant because if nodes have to wait for acknowledgments for these messages, the total migration time then depends on the distance between nodes. When the cost of migration is defined in terms of migration time, only the time taken to move the state is generally included in the equation, and might result in an inaccurate view of the cost.

The bandwidth delay product is a measure of how much data can be sent in a given duration. As part of the cost of migration, it represents the amount of data that can be sent when a migration is underway. The more tuples that can be sent, the higher is the cost of migration, and the less desirable a migration is. MigCEP [96], [98] uses the average bandwidth delay product during migration as its cost. This represents the utilization of the network due to migration.

The monetary costs of migration have been modeled by Zacheilas et al. [60] and Hiessl et al. [90]. VISP [90] distinguishes between the enactment cost of a placement, which is the cost of running the current topology, and the migration cost, which is the cost of making a change.

Enactment cost:

$$C_{op}(x) = \sum_{i \in V_{dsp}} \sum_{u \in V_{\mathrm{res}}^{\mathrm{i}}} C_u x_{i,u} \quad (4)$$

Migration cost:

$$C_{mig}(x) = \sum_{i \in V_{dsp}} \sum_{u \in V_{\mathrm{res}}^{\mathrm{i}}} \sum_{u \in V_{\mathrm{res}}^{\mathrm{i}}} C(i,u,v)\ x_{\mathrm{i,u}}^{\mathrm{prev}} x_{i,v} \quad (5)$$

where V_{dsp} represents the operators, V_{res} represents the compute nodes, $x_{i,v}$ represents the placement of operator $i \in V_{dsp}$ on the new host $v \in V_{res}$, and $x_{i,u}^{prev}$ represents the placement of operator $i \in V_{dsp}$ on the old host $u \in V_{res}$.

Considering both the enactment cost and the migration cost makes it possible to assess whether the long-term cost savings from scaling down to fewer instances exceed the short-term cost of migration, leading to a reduction in the frequency of migrations.

D. Benefit

To explore the optimization goals of different migration solutions, we begin by examining the most important metrics used to assess the benefits of migration. The benefit of migration is based on performance in terms of the placement, amortization time, and the cost of migration (as explained in Section III-B). This is either explicitly defined or implicit in the decision-making, where the goal is to maximize performance in terms of the placement and minimize the cost of migration.

One could argue that all goals of optimization are relevant to all goals of migration. However, some are more tightly coupled than others. For instance, load balancing involves using the load of a system to make balancing decisions. QoS solutions, on the contrary, are not bound specifically to any goal of optimization. Elasticity-based solutions aim to minimize resource usage while maintaining the QoS. In other words, they use as few resources as possible for an application, and trigger a scaling operation when the load is above or below a given threshold. Fault tolerance-based solutions involve migrations when nodes fail and the operators must be migrated to new or existing nodes.

Network performance as a goal of optimization means using the quality of the network links to determine performance in terms of placement. Important metrics in this context include the bandwidth between links in the overlay topology, the latency between nodes, and the bandwidth delay product. Tuple processing performance in query processing is the most popular indicator of the quality of an adaptation, as shown by the number of studies that have measured the benefit of migration in terms of tuple latency or rate. If a node is overloaded in a data center, the latency of the tuple might exceed acceptable levels, leading to QoS violations. A long migration time might temporarily worsen performance, but if the general gain in performance outweighs the degradation, the migration is considered worth it. The load of a system is an important goal of optimization that makes it possible to run as many operators on a node as it can handle, and to make changes when the workload is above or below a given threshold. The cost of migration is essential to consider when making migration decisions to avoid excessively frequent migrations and ensure that the benefit of the new placement outweighs the cost of migration. When the cost of migration is used to calculate its benefit, the result is the modeled benefit of migration. Monetary cost can be useful as a goal of optimization to make a tradeoff between the cost of resources and the performance of the system.

1) Network: In decentralized fog and edge computing solutions, network usage as well as bandwidth and latency between links are crucial metrics. Pietzuch et al. [117] developed an overlay network that can make network-aware placement and migration decisions. Parameters, like the latency and bandwidth of overlay links and the load on nodes are used as criteria of optimization when placing and migrating operators. Rizou et al. [88] implemented a similar method that converges to the optimal placement in fewer migrations than the solution by Pietzuch et al. [117].

2) Tuple Performance: Being able to analyze streaming data as soon as it arrives and to react immediately to certain patterns in the data is one of the core motivations for SPEs. Therefore, tuple performance is of significant importance. Furthermore, in a resource-constrained environment, tuple latency can be an indicator of energy consumption and the goal to minimize latency can implicitly lead to energy reduction. For most existing approaches, the goal of migration directly or indirectly involves improving performance. Most elasticity-based and load balancing-based solutions are cluster-based, and are more concerned with the load on the system than the bandwidth of or latency between links.

The tuple rate of a data stream can be used to detect backpressure, i.e., when the input rate is higher than the processing rate. This can be used as an indicator of the load on the nodes, and to calculate the variance in load. For instance, Buddhika et al. [66] proposed a methodology to reduce the interference between stream processing operators using migration. To achieve this, the interference score of an operator is calculated, where the higher the score is, the greater the need is for migration. This interference score is based on the prediction of future packet load. Similarly, the WASP system [92] relies on the expected input and output rates of an operator instead of merely on the observed rates. Repantis and Kalogeraki [86] defined latency constraints on the operators and used tuple latency to determine when an operator must be migrated.

Tuple latency is a common constraint to have on the operator performance. If the latency constraints are not kept, it causes the system to scale out or perform load balancing. Röger et al. [91] studied the relationship between latency constraints and monetary costs. In particular, the lower the latency constraints are, the more instances a system needs to run, and thus, the more costly the operation is. As such, the optimization problem is:

$$\begin{aligned} &\min && \sum_{cu \in path} cost(cu, latency) \\ &\forall paths && \sum_{cu \in path} latency(cu) <= \text{e-to-e latency bound} \end{aligned} \tag{6}$$

3) Load: Unsurprisingly, all load balancing solutions use either load as a parameter when making decisions or tuple performance to model load. One method is to minimize the variance in load between nodes in a cluster [57]. In this case, a coordinator monitors the load on the system nodes and, when a balancing decision has to be made, selects the configuration with the least variation in load. This might, however, require expensive migrations of large loads among many nodes, and redistributing loads that are not the cause of the imbalance. Another method is to trigger a load balance when the imbalance has crossed over a given threshold to re-balance the load to at least below a given threshold [57]. In other words, load balancing is used as a constraint. In this case, the goal is to minimize the cost of migration by redistributing the minimum amount of load to achieve an acceptable load balance. This method achieves an acceptable load balance while moving the smallest load.

Resource usage can mean multiple metrics that relate to the system's workload. It can be the number of threads assigned to the operators, as described in the work by Xu and Palanisamy [100]. Alternatively, it can also be understood in terms of the CPU queue state of a computing node, as illustrated by Sun et al. [77]:

$$l_{cn,[t_s,t_e]} = \sum_{v_i \in V_{cn,[t_s,t_e]}} n_{v_i,cn,[t_s,t_e]}, \tag{7}$$

where $n_{v_i,cn,[t_s,t_e]}$, indicates the number of tuples of operator v_i during $[t_s, t_e]$, and $v_i \in V_{cn,[t_s,t_e]}$. The load is then partially based on the tuple rate, and later used as a constraint in a minmax load balancing optimization problem.

Elasticity-based solutions increase or reduce the number of resources used by an application based on its variable workload. If a cluster is overloaded after load balancing, this is a sign that the system should scale out [104]. In decentralized fog-based solutions, the load of a system is not known beforehand, and therefore, there might be a tradeoff between latency and load. Pietzuch et al. [117] introduced a cost space model in which a topology of systems is constructed based on the latency and bandwidth between nodes as well as the load on systems. If the load of a system is large, the relevant node appears farther in the cost space when mapping an operator graph to a physical topology, and thus is less likely to be selected.

4) Monetary Costs: In the cloud model, followed by the fog and edge models, users mostly pay based on usage. Users can allocate a certain amount of resources and scale out or in whenever more or less resources are needed, to keep the resource usage cost low. A complicated issue in this case is balancing the monetary costs with the benefits of improved placement. None of the load balancing solutions use monetary cost as an optimization criterion. This makes sense as the load balancing problem involves evenly distributing the load over a fixed amount of resources, whereas elasticity can increase or reduce the amount of resources. In terms of hardware resources, there is nothing to optimize as they are already paid for. The monetary cost of moving states during migration can thus be minimized. Typically, this is implicitly done by designing the objective function to minimize the number of state that need to be moved. Elasticity-based solutions require a tradeoff between resource usage and monetary costs [60], [65], [89]. An elastic solution might use a threshold for the load to determine when to scale out. However, deciding when to scale in might be more complex, considering that it requires a certain downtime for the worker to be removed.

5) Costs of Migration: Any type of migration introduces some costs. However, this does not mean that a migration always affects the QoS. If no tuples arrive during the downtime of the operator no tuples will be affected and neither the QoS. Most studies aim to prevent the cost of migration from affecting the QoS by implicitly minimizing the number of migrations, their frequency, or their magnitude. Zhou et al. [49] emphasized the need to minimize the time needed for query migration but did not describe a means of implementing this in their solution. Lombardi et al. [68] defined the cost of migration in terms of the time it takes to perform different steps but did not attempt to minimize it. The cost of migration can be minimized by either using single-objective optimization [57], [73], [76], [92], or simple additive weighting (SAW) with multiple objectives [50], [52], [72], [89], [95]. If only the cost of migration is minimized, constraints have to be placed on the quality of the placement to ensure that the selected placement is acceptable. With load balancing, minimizing the cost of migration while maintaining constraints on the load imbalance is a good way to ensure a balanced load that minimally affects the performance of the system.

Minimizing the number of migrations is a similar goal to minimizing the cost of migration. Repantis and Kalogeraki [86] proposed a hotspot alleviation-based solution with the goal of minimizing the number of migrations that leads to an acceptable QoS for the operators. Rizou et al. [88] implemented a similar relaxation algorithm to the one in [117], and showed that it requires fewer migrations before converging to the optimal placement and fewer control messages. The easiest way to prevent needless migrations is to use a threshold that ensures that they are beneficial. Load balancing systems commonly use thresholds of load imbalance to ensure that the load is redistributed only when the load imbalance is above a certain threshold. A different type of threshold targets the migration itself to ensure that its benefit is worth its cost. Pietzuch et al. [117] proposed a method that migrates data only when the benefit in terms of network capacity is higher than a threshold based on the cost of migration.

Using the cost of migration as a goal of optimization means penalizing a placement alternative based on it. Even if a placement is better than the given placement, it might not be preferred because the cost of the reconfiguration is too high. In load balancing-based approaches, the cost of migration is commonly minimized but most often as an implicit goal rather than as part of the objective function. The goal is generally to achieve an acceptable load distribution as quickly as possible, and the redistribution itself constitutes the highest cost. The cost of migration can be minimized while maintaining a balanced load [73]. The number of migrations can be minimized while fulfilling QoS requirements [86]. Another way is to maximize the improvement in a query plan and divide the improvement in performance by the cost of migration [52]. Load balancing decisions can be made with cost of migration in mind in multiple ways [57]: minimizing the cost of migration with load balancing as a constraint, keeping the cost of migration as a constraint while minimizing the load imbalance, or combining load and cost of migration to minimize both.

In elasticity-based approaches, the cost of migration is often considered in the same way as in load balancing because scaling can be considered to be an extension of load balancing. Zacheilas et al. [60] minimized the monetary costs of computational resources, the cost of migration, and the cost of missing tuples. In this approach, a tradeoff is made between the cost of resources, the cost of missing tuples, and the migration time. A reinforcement learning-based approach was used in [89] that minimizes the cost of reconfiguration, the performance penalty due to QoS constraints, and the cost of resources for using the computational resources.

E. Proactive Migration Decisions

Current migration solutions generally use reactive approaches to make migration decisions. That means migration is a reaction to a certain trigger-event that happened. For instance, a migration might be triggered if a node is overloaded and QoS guarantees are violated, such as when the tuple latency increases excessively. In contrast, proactive migration is performed before a trigger-event happens. This means that the trigger-event needs to be predicted, typically based on historical monitoring data. Most proactive solutions predict whether the node can sustain the workload.

In Table XII, we summarize the studied papers that contribute with proactive decision-making strategies for operator migration. For each paper, we highlight the specific prediction approach and its key idea.

There are many ways to model or estimate the metrics described above. The classical way is to collect some measurements from possible migration hosts and formulate an optimization problem using, e.g., ILP, and then attempt to solve it. This is done by Cardellini et al. in [72], [138].

Some solutions predict the adaptability of QoS violations [86], [104]. Repantis et al. [86] used linear regression and the incoming tuple rate to predict QoS violations of the end-to-end execution time. They predicted QoS violations to prevent them. Lohrmann et al. [104] built a predictive latency model using queuing models and Kingman's formula [153] to make scaling decisions.

Zacheilas et al. [60] estimated the load and expected latency of Esper to make scaling decisions by using Gaussian processes [154] because they can help to estimate the uncertainty in predictions. However, this method has a cubic computational complexity due to the use of matrix inversion. Liu et al. [82] used the extended Gaussian Processes upper confidence bound algorithm to search for the optimal configuration for bottleneck operators for modeling the service capacity. Wang et al. [69] predicted resource usage in real time to choose the configuration that can minimize CPU and memory resources while fulfilling QoS guarantees. This is done using incremental learning techniques based on Weka [155] and MOA [156]. De Matteis and Mencagli [107] used MPC to predict optimal scaling decisions, called the future horizon. Buddhika et al. [66] used prediction rings to forecast the interference score that expresses the degree to which a system is expected to be overloaded. Lombardi et al. [68] used a reactive and a proactive mode for making scaling decisions in their Elysium system. In the reactive mode, the tuple rate is used as the basis for decisions, and in the proactive mode, the input load is predicted over a certain time, called the prediction horizon.

In [89], a reinforcement learning approach is applied to decide when to perform scaling operations. Liu et al. [74] predicted the load of operators as the number of tuples that operators need to process during a prediction horizon. In WASP [92], the expected input and output rates of the operators are estimated as an alternative to backpressure monitoring for estimating load. Backpressure is weaker as it is based on the observed load instead of the actual workload, and this may lead to less accurate adaptation decisions [150]. A composition of reactive, proactive, and delayed migrations was presented in [152]. The results of this empirical study indicated that knowledge of the window state can be used to schedule a migration when the state is minimal (i.e., after completing a tumbling window, as in [99]), or when no output tuple is affected by the migration.

The Phoebe system [81] predicts future workloads as a way to make near-optimal scaling decisions. This is done using models for predicting the end-to-end latency of tuples and recovery time of the system. The end-to-end latency model uses multiple regression and clustering for estimating latencies of scaleouts and workload rates. The recovery time model bases its predictions on multistep-ahead time series forecasting [157] of the expected workload rate over time and a regression model for predicting the maximum processing capacity of the system.

VI. Empirical Quantification of Core Concepts of Migration

The previous two sections have given the reader an understanding of how operator migration works and which design choices for migration mechanisms and decisions have been investigated in existing works. The aim of this section is to complement the understanding of the functional aspects of operator migration with some insights into the impact of design decisions on the performance of operator migration. Therefore, we first define two direct moving state migration algorithms: (1) one that uses partial state movement, and (2) another that sends the entire state at once. They are defined in an abstract way such that they can be implemented in different SPEs, and we have decided to implement them in the two popular SPEs Apache Flink and Siddhi. We define decision models to determine when and where to migrate the data. We conducted a real migration experiment to analyze the migration algorithms based on the NEXMark benchmark [10]. We show a use case of the decision models for migration to illustrate their effect on decision-making.

A. Migration Algorithms

The difference between the partial state movement algorithm and the all-at-once state movement algorithm is that the former splits the state into a large static state and a small dynamic state. The static state is transmitted while the operator is still running and processing tuples, followed

TABLE XII
PROACTIVE MIGRATION PREDICTION TECHNIQUES

Paper Reference	Prediction Approach	Key Idea
Repantis et al. [86]	Linear Regression	Uses incoming tuple rate to predict QoS violations
Lohrmann et al. [104]	Queuing Models	Develops a predictive latency model for scaling decisions using queuing models and Kingman's formula
Zacheilas et al. [60]	Gaussian Processes	Estimates load and expected latency for scaling decisions
Liu et al. [82]	Extended Gaussian Processes	Uses the upper confidence bound algorithm to search for optimal configuration for bottleneck operators
Wang et al. [69]	Incremental Learning	Predicts resource usage in real-time to choose a configuration that minimizes CPU and memory resources
De Matteis et al. [107]	Model Predictive Control (MPC)	Predicts optimal scaling decisions
Buddhika et al. [66]	Prediction Rings	Forecasts the interference score to predict system overload
Lombardi et al. [68]	Tuple Rate	A proactive scaling mode that predicts the input load over a specific horizon to make scaling decisions.
Cardellini et al. [89]	Reinforcement Learning	Applies reinforcement learning to decide when to perform scaling operations
Liu et al. [74]	Load Prediction	Predicts the load of operators as the number of tuples they need to process during a prediction horizon
Jonathan et al. [92]	Operator Input and Output Rate Estimation	Uses expected input and output rates for load estimation
Lindeberg et al. [152]	Window State Monitoring	Uses knowledge of the window state for migration scheduling
Geldenhuys et al. [81]	Multiple Regression and Time Series Forecasting	Predicts future workloads for near-optimal scaling decisions

by the extraction of the dynamic state. As such, static state transmission involves little or no overhead in query processing, and constitutes only one additional step in the algorithm. Note that a partial state movement algorithm might split the state into more than two parts, such as in Megaphone [75] and Meces [80].

We use the algorithms described in Section IV as basis. In particular, we divide the algorithms into functions that are executed by different nodes participating in the network according to their roles. When moving the state, the old host provides the next hops for the query. Thus, there is no need to add them explicitly in these tasks. These tasks follow a similar format to that used in Expose [123], which is a framework and toolset for efficiently defining and executing DSP experiments. Wrappers for different SPEs are provided such that all SPEs support a common set of tasks. Expose has been extended with additional tasks to enable operator migration.

Listings 6 and 7 describe the tasks we use to define the all-at-once state movement algorithm and the partial state movement algorithm, respectively. They differ slightly from similar algorithms in Section IV in some respects, such as the ways in which streams are managed. It is possible to send a batch of tasks to upstream nodes, as in Flux [46], to the new host as in [50], and to the old host as in [85]. The difference between the all-at-once state movement and partial state movement algorithms is that the latter involves sending the current state of the query before the operator is paused, achieving the same effect as in checkpoint-assisted solutions.

Implementation: To facilitate the migration of any moving state operator, an SPE needs to be able to extract the runtime state and the load state. This feature is supported in different ways by Siddhi and Flink. In Siddhi, the state is loaded from the runtime system into a byte array, and requires that the entire state is available in memory. As such, there are limitations on how large the state can be. On the contrary, Flink writes the state as a set of checkpoint files, each of which does not exceed a configurable size. Therefore, the state to migrate with Flink can be larger than in Siddhi. The implementation of the other tasks, including `BufferStreams`, `StopStreams`, `ControlMessage`, `AddNextHop` and the rest, is supported through simple tasks defined in the SPE wrapper in Expose [123].

The standard moving state algorithm is implemented in Flink and Siddhi, but only Flink supports partial state movement since this requires the ability to split a given state into a large, immutable state and smaller incremental checkpoints. This feature is supported by one of the state backends in Flink called RocksDB [158]. Flink with RocksDB is also used for the checkpoint-assisted algorithm in Rhino [78], which uses partial state movement. Another benefit of RocksDB is that it does not store the entire state in memory while the system is running, but instead writes it to file and minimizes its size based on multiple criteria.

As discussed in Section V, there are many different ways of making the migration decision. Our solution is to make the decision process as transparent and meaningful as possible by optimizing the QoS. The goal is to maximize the performance of a placement while penalizing it based on the cost of migration, which varies for different nodes and is zero for the current host. In this way, it is clear why a new placement is selected over the old one, for reasons other than simply that the old host is over-provisioned or the new placement delivers better performance.

The amortization time (*at*) varies depending on the reliability of a placement score for the operator on a given host. If the placement score is stable over time, the amortization time increases since it is less likely that the placement becomes suboptimal shortly after the migration. For instance, a mobile node might have less consistent placement score than a server located in a data center, and as such, it is even more important that the migration is worth the cost of it.

$$at(h, op) = min_{at} + (max_{at} - min_{at})/100 * (100 - rsd_p(h, op)) \tag{8}$$

```
ControlMessage(OH
    ControlMessage(NH,
        RequestMigration(query),
        BufferStreams(Streams(query))
        StopStreams(Streams(query)))
    ControlMessage(Upstream,
        Redirect(Streams(query), OH, NH))
    MoveState(query, NH)
    AddNextHop(Streams(query), NH))
```

Listing 6. All-at-once state movement.

```
ControlMessage(OH
    ControlMessage(NH,
        RequestMigration(query),
        BufferStreams(Streams(query))
        StopStreams(Streams(query)))
    MoveImmutableState(query, NH)
    ControlMessage(Upstream,
        Redirect(Streams(query), OH, NH))
    MoveIncrementalState(query, NH)
    AddNextHop(Streams(query), NH))
```

Listing 7. Partial state movement.

where $rsd_p(h, op)$ expresses the relative standard deviation (RSD) of the historical placement scores of operator *op* on host *h*.

When defining the cost of migration, operator downtime alone is not sufficient, because it does not reveal how many tuples, if any, are affected by the downtime. Therefore, we use the tuple rate during the migration as a foundation for the cost of migration. Since the data sink waits for tuples from the operator, we consider the number of expected output tuples $PT_{out}(at, op)$ that are affected by the migration to calculate its cost. Buddhika et al. [66], Phoebe [81] and Liu et al. [82] describe tuple prediction methods that can be applied here:

$$PT_{out}(at, op) = PT_{in}(at, op) * Sel(op) \tag{9}$$

where $PT_{in}(at, op)$ is the predicted number of input tuples for operator *op* during amortization time *at* and *Sel*(*op*) is the selectivity of operator *op*, which could for instance be a join, pattern matching operator or an aggregation operator.

The cost of migration can be calculated as the operator downtime divided by the amortization time. Since we focus on output tuples from the query, the cost of migration *C*(*op*, *oh*, *nh*) is defined as the ratio of the predicted output tuples ($PT_{out}(mt(oh, nh, op))$ from a query during migration to the output tuples predicted from it during the amortization time ($PT_{out}(at(nh, op))$).

$$C(op, oh, nh) = w_c * \frac{PT_{out}(mt(oh, nh, op))}{PT_{out}(at(nh, op))} \tag{10}$$

The cost has a weight associated with it, meaning that the system can dynamically change how much the cost of migration matters based on the selected policy. If w_c is set to one, this suggests that the performance of a placement should be reduced in proportion to the number of tuples that are received during operator downtime. If w_c is set to 1.5, the placement is penalized further. This makes sense as buffered tuples may take some time to process, during which time no new tuples may be processed.

B. Decision Models

Given the amortization time, the benefit of the migration $B_m(op, oh, nh)$ of a placement is its finite performance penalized by the cost of migration, instead of it being a general placement score. Of two placements with the same migration cost, the one with the higher placement score is selected. The only difference arises when two placements have different costs of migration, for instance, when comparing the given placement with zero cost of migration with another placement that requires a migration. The benefit of migration can be calculated as:

$$B_m(op, oh, nh) = P(nh, op) * (1 - C(op, oh, nh)) \tag{11}$$

where *P*(*nh*, *op*) is the estimated placement score for the new host *nh* running operator *op*.

The above functions show how the migration decisions are made. Migration checks are periodically performed by calculating the placement score. Following this, the benefit of the migration of placements is calculated by penalizing the placement score based on the cost of migration. We define *M*(*oh*, *phs*, *op*) as the potential host with the maximum benefit for the given operator. This host is selected as the future host for the operator, and triggers migration if it is not the given placement:

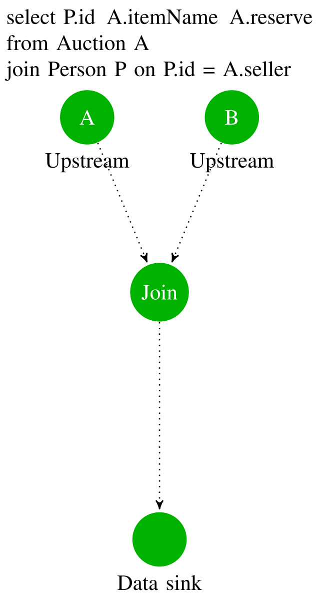
$$M(oh, phs, op) = \max_{ph \in phs} B(oh, ph, op). \tag{12}$$

C. Empirical Evaluation

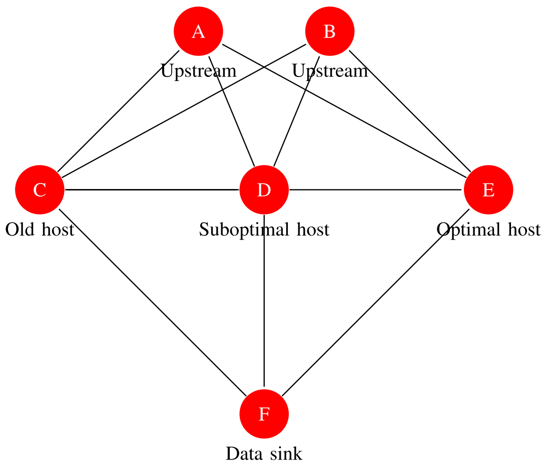
We quantitatively analyzed the proposed decision models for migration through a use case and our migration algorithm through experiments. The goal was to show the usefulness of incorporating the cost of migration into the process. We considered a use case for the decision models, because it makes the analysis and discussion of the results easier. On the other hand, implementing and running the migration algorithms on SPEs is necessary to understand the impact of migration.

Figure 14 illustrates our evaluation scenario: Figure 14a shows the operator graph used for both the use case and the migration experiment, and Figure 14b shows the DSP overlay topology. The mapping from the operator graph to the physical topology is demonstrated using the decision models in Section VI-C1, and an experiment involving the migration of state from the join operator on one node to another is described in Section VI-C2.

1) Decision Model Use Case: The decision models for migration were assessed in this use case. They were applied using a prediction model oracle with 100% accuracy to make migration decisions. We expect that the migration time can be predicted based on periodically updated topological information and network statistics. By using knowledge of the number of tuples sent in the time window and the migration time, we can predict the total end-to-end latency of the tuples during a given time window. The parameters of the use case


select P.id A.itemName A.reserve
from Auction A
join Person P on P.id = A.seller
A
Upstream
B
Upstream
Join
Data sink

(a) Operator graph



(b) DSP overlay topology

Fig. 14. Evaluation scenario.

TABLE XIII
PARAMETERS OF THE USE CASE

Parameter name	Parameter value
Amortization time	5 s
Bandwidth C< − >D	200 mbit/s
Bandwidth C< − >E	100 mbit/s
Bandwidth D< − >E	100 mbit/s
Bandwidth Leader< − >Hosts	200 mbit/s
Latency between all links	1 ms
Control message size	168 bytes
Migration cost (C)	0
Migration cost (D)	0.1
Migration cost (E)	0.5

TABLE XIV
PLACEMENT SCORE AND AMORTIZATION TIME PARAMETERS FOR EACH SPECIFIC RUN IN THE USE CASE

Run	at	P (C)	P (D)	P (E)
1	1000	1.5	1	2.7
2	2000	1.6	1.6	2.5
3	3000	1.4	2.5	3
4	4000	1.7	2.8	2.9

TABLE XV
RESULTS OF THE USE CASE

Run	B_m (C)	B_m (D)	B_m (E)	M (CM)	M (NCM)
1	1.5	0.85	1.35	C	E
2	1.6	1.36	1.25	C	E
3	1.4	2.125	1.5	D	E
4	1.7	2.38	1.45	D	E

are provided in Table XIII. We considered two source nodes A and B, three potential hosts C, D, and E, and a sink node F.

Results: Table XV shows the results of the use case for the configurations given in Table XIII and the run-specific parameters are provided in Table XIV, including amortization time and placement scores for the potential hosts. These scores would in a real-world system be calculated based on the expected performance of a node that runs the system. Node E has the best P score in all runs, as illustrated in Figure 14, which means it is the best candidate for running the operator. However, the cost of migration, as indicated in Table XIII, shows that Node E has five times higher migration cost than migration to Node D, which leads to Node E only being preferred when the cost is ignored (M (NCM)), and Node C and D are preferred when the cost is considered (M (CM)). The previously described equations are used along with the *at* and P scores to define the benefits of the potential placements, and the preferred host is selected based on M.

Since the P score was stable for Node E, and was significantly better than that for Node C, a decision policy may decide to dynamically increase the amortization time for nodes that had demonstrated their stability in terms of the predicted P score. On the contrary, Node D has a significantly variable P score, which increased above that of Node C with a value of 1.6 in the second row, but yielded a lower benefit of migration of 1.36, and was not selected as the new host. With a score of 2.5 that was reduced to 2.125 given the migration cost, it beat the given host, and was selected as the new host.

2) Migration Experiment: In this experiment, we demonstrated two migration algorithms by analyzing and comparing their execution results in two different SPEs, with one SPE limited to a single algorithm. The all-at-once state movement runs were used to send 100,000, 1,000,000, and 5,000,000 tuples. The partial state movement runs were used to send between 1,100,000 and 300,100,000 tuples. The noticeable differences in the number of migrated tuples between the all-at-once and partial state movement runs were a result of the limitations in the SPE's state backends, as explained in further detail below. The additional 100,000 tuples with partial state movement were sent during migration, and were part of the

TABLE XVI
SERVER SPECIFICATION

OS	CPU	RAM	Description
Ubuntu 20.04.2	Intel Xeon Gold 5215 2.50 GHz	125 GB	Old host
Ubuntu 18.04.4	Intel Core i7-7800X 3.50 GHz	377 GB	New host, data producer, data sink, Expose coordinator

TABLE XVII
RESULTS OF ALL-AT-ONCE MOVING STATE EXPERIMENT

SPE	# Tuples	State size	State extraction	State loading	State transfer	Freeze time
Siddhi	100,000	100 MB	2.8 s	1.76 s	0.94 s	3.6 s
Siddhi	1,000,000	1 GB	21.3 s	13.6 s	9 s	43.9 s
Flink	100,000	100 MB	270 ms	2.3 s	1.1 s	3.6 s
Flink	1,000,000	1 GB	3.3 s	20.3 s	11.6 s	35.2 s
Flink	5,000,000	5 GB	19.8 s	77.4 s	52.8 s	150 s

TABLE XVIII
PARTIAL MOVING STATE EXPERIMENT RESULTS

SPE	Tuple count (S)	Tuple count (D)	Size (S)	Size (D)	State extraction (S)	State extraction (D)	State loading	Transfer (S)	Transfer (D)	Freeze time
Flink	1,000,000	100,000	1 GB	113 MB	76 ms	149 ms	1.77 s	11.6 s	111 ms	2 s
Flink	5,000,000	100,000	5.2 GB	376 MB	363 ms	528 ms	3 s	57.2 s	4.3 s	7.8 s
Flink	25,000,000	100,000	26.1 GB	3 GB	23.3 s	1 s	8.6 s	4.35 min	30 s	39.6 s
Flink	50,000,000	100,000	52.2 GB	2.74 GB	15.4 s	1.2 s	16.8 s	8.67 min	27 s	45 s
Flink	100,000,000	100,000	104.5 GB	239 MB	618 ms	911 ms	63.8 s	17.47 min	2.5 s	67.2 s
Flink	200,000,000	100,000	208.8 GB	279 MB	61.2 s	18 s	7.7 min	35.1 min	2.6 s	8 min
Flink	300,000,000	100,000	313.3 GB	4.9 GB	31.7 s	16 s	13.1 min	52.5 min	53.3 s	14.3 min

dynamic state to be sent. The experiment used a simplified version of the topology in Figure 14b in two ways. First, there was only one upstream node. Second, there were only two hosts: the old and the new host.

The experiment tested the cost of migration by varying the size of the state to be moved. For runs of the partial state movement, the number of tuples that were migrated during static state migration and dynamic state migration were varied. The dataset of the NEXMark stream processing benchmark [10] was used in the experiment. NEXMark is based on an auction scenario, where three streams are used: a Person, a Bid, and an Auction item stream. For this experiment, only one of the queries was used, one that joined the Person and Bid streams. We used this query because a join query makes it easier to test the migration algorithm and adjust the size of the state to migrate. One can simply send a given number of tuples of the first stream, migrate it to the new host, and send a single tuple of the second stream to the new host. If this triggered the correct number of output tuples to be produced, the migration was considered to have been successful.

Four processes with different roles were used in the experiment: a data producer node, an operator host running the operator to be migrated, a new host that contained the operator after migration, and the data sink that consumed the output tuples of the operator. We used two machines for the experiment, one for the old host, and the other to run the data producer, data consumer, and new host. The machines were connected via Ethernet cable in a local area network. The specifications of the machines are shown in Table XVI. In the experiment, the data producer generated a certain amount of Auction tuples that were sent to the old host. The state was then migrated to the new host, and the data producer sent a single Person tuple that joined with all the Auction tuples to trigger the same number of output tuples to be sent to the data sink as Auction tuples that were sent prior to the migration. The query we used was a modification of NEXMark's [10] Query 8. Originally, this query does not select the itemName of the auction, but chooses the person's name. Each Auction tuple was augmented with 1 kB of a randomized string to increase the size of the state to be migrated.

In all runs, we counted the number of tuples that were migrated, the state size, the state extraction time, the state transfer time, and the state loading time. For the partial state movement algorithm, the same parameters were used for the static and dynamic states. The state to be migrated ranged from 1 to 300 GB. However, Siddhi has a limit of 1 GB because it extracts the entire state into a single byte array, whereas Flink's state backend RocksDB splits the state into multiple files. RocksDB is used in both the all-at-once approach and partial state migration approach, but where all-at-once disables incremental checkpointing and partial state migration enables it. When migrating all-at-once with Flink, the maximum state that could be migrated is 5 GB, because the checkpoints fail at larger states. It is unknown why this issue occurs. It would be possible to run Flink with all-at-once migration with incremental checkpoints, but then it would be the same as the partial state migration, except where the state transfer of the static state is added to the freeze time.

Results: Tables XVII and XVIII show the experimental results of the all-at-once state movement algorithm and the partial state movement algorithm, respectively. Siddhi and Flink migrated operator states of different sizes depending on the query and the number of tuples that were processed.

The state transfer times of Siddhi and Flink were similar because they used similar implementations of the TCP socket. Siddhi performed slightly better, because Flink had to read the checkpoint from multiple files, and state transfer was executed in parallel with reading the files. State extraction appeared to scale relatively poorly for both Siddhi and Flink with the all-at-once state movement algorithm, but with the partial moving state, Flink had a significantly lower state extraction overhead. Moreover, state loading using partial state movement was much faster than without it. Note that these results do not represent the general performance of the SPEs, but the outcomes for a specific join query that was used for a specific system. Another query might have yielded different results. For instance, this query was very write heavy and the Auction tuples were made to be larger in size than the benchmark normally defines. In this case, the partial state movement algorithm performed better in all respects.

One might think that the all-at-once state movement algorithm would have had faster state loading as it has a monolithic checkpoint, but this was not the case. We think this result is obtained because the incremental checkpointing uses RocksDBs native checkpoint files whereas Flink's full snapshot approach iterates through the RocksDB state and creates its own files. RocksDB is designed to be efficient, and performs indexing to increase its efficiency. This benefit was lost in the full snapshot approach.

If we assume that the number of tuples that were received during the freeze time arrived at a fixed rate, the average additional tuple latency as a result of the migration would be equal to half the freeze time. The maximum additional tuple latency would be approximately equal to the freeze time and the minimum was close to zero. The number of affected tuples could vary significantly, ranging from zero to hundreds of thousands per second.

The partial state movement algorithm performed much better than the all-at-once algorithm in terms of freeze time, almost 20 times less freeze time for the partial state movement algorithm versus the all-at-once movement algorithm when the state to migrate was around 5 GB. There are two reasons for the performance gain. First, using the incremental checkpointing led to lower state loading times. Second, most of the state was moved before the operator was shut down. This difference in performance was especially significant when considering how similar the algorithms were in terms of how they were described in Listings 6 and 7. Only one task was added to Listing 7, which was to migrate the immutable state before the streams were redirected by the upstream nodes. This leads to the important conclusion that the literature can benefit from a common language when defining or using a migration algorithm. Exactly what tasks are executed during the migration, in particular, those that increase the freeze time, can be described using, e.g., the concepts of the migration model in Section III.

Table XVIII shows that the size of the dynamic state in the partial state movement runs was unpredictable. For instance, when 25 million tuples were migrated, the size of the dynamic checkpoint containing 100k tuples was 3 GB, whereas it was only 279 MB for 200 million tuples. However, the actual size of the 100k tuples remained around 100 MB across all runs. This could be attributed to the fact that we disabled RocksDB compaction after extracting the static state. Had we kept the compaction enabled, the final incremental checkpoint would have potentially merged with the static state, resulting in a new set of state files that would be incompatible with the ones sent to the new host.

VII. Reflections and Future Directions

The historical development in operator migration, from the early single-track moving all-at-once state migration solutions to checkpoint-assisted partial state movement and parallel-track solutions without state movement, has been driven by the deployment of SPEs to the cloud environment, and improvements to them to achieve fault tolerance and dynamic scalability. The core ideas to achieve this are related to resource availability and state management. Cloud environments provide large amounts of computational resources (even though at different scales), and their servers are interconnected with low-latency high-bandwidth networks. This allows to execute operators in parallel to improve migration performance at the cost of higher resource utilization. Early single-track migration solutions treat the state as a single large binary object as such there is no other way to migrate the entire state all-at-once.

More advanced state management solutions allow to partition the state which in turn leads to more design options for migration solutions. Checkpointing, distinguishing between immutable and mutable state, and prioritization of state partitions are examples for partition mechanisms. The more advanced solutions, e.g., based on prioritization, consider the semantics of the state to determine which piece of the state should be migrated first to improve the migration performance. Very recent approaches follow the idea of prioritization to perform state shedding, which reduces the size of the state to be migrated at the cost of inconsistency between the state at the old and new host. Another way to reduce the state size is to schedule the migration. Based on these insights on how to improve operator migration for cloud-based environments it is reasonable to expect that such solutions might also work well in fog environments.

However, in fog environments that are geo-distributed, the connections between hosts have substantially lower available bandwidth and higher latencies that can impact the cost and benefit of operator migration, and require adapted migration mechanisms. The periodic checkpointing and replication of checkpoints are used in some cluster-based SPEs to facilitate fault tolerance and fast migrations, but it is not always feasible to replicate and distribute checkpoints, especially in resource constrained IoT devices. For future in-network processing solutions with mobile platforms, e.g., advanced crowd-sensing.

It is clear that the smaller the size of the data to be migrated is, the less energy is consumed. Therefore, scheduling operator migration at a point in time when the state is small or even zero is important. This can be achieved, for example, by delayed migration by waiting until a tumbling window is emptied [98], and through proactive migration. Another alternative is to allow for some inconsistent state, i.e., not the entire state is migrated to the new host. In some cases, aggregation operators can be moved without the state, resulting in zero freeze time. Alternatively, load shedding techniques can be applied to send some of the state, or components of it can be assigned a priority such that only the most important state is migrated, while the less significant part of it is omitted. However, a thorough investigation of the pros and cons of reactive, delayed, and proactive migrations in different environments with different workloads and guarantees of consistency is still elusive.

Another gap in research is an analysis and comparison of stream management techniques. Several aspects are important for such an investigation: (1) the sequence of tasks like the stopping, buffering, redirecting, and starting of streams, (2) the locations where streams are buffered, (3) the delivery semantics, i.e., at least once, at most once, exactly once as well as ordered or out-of-order delivery, and (4) tasks related to buffer management and transport protocols.

The quality of decision-making on migration depends on the data available to calculate its cost and benefit, as well as the freshness of the data. The continuous collection and dissemination of monitoring data in DSP can be expensive. Efficient monitoring solutions, and leveraging other sources of monitoring data that are, for example, used for network and system management have the potential to reduce the overall cost of DSP and ensure good decision-making.

Leveraging historical data to perform predictions with advanced statistics or modern machine learning solutions, as is done for traffic prediction in network management [159] and data prediction in wireless sensor networks [160], is another subject that deserves more attention in research. Some studies have already explored proactive migration techniques, as discussed earlier in the paper, but further investigation into this area remains necessary. Both proactive migration and the use of amortization time in the cost model require some form of prediction. The oxymoron of operator migration, i.e., that the need for migration occurs when the cost of migration is high, can be avoided with proactive migration. Furthermore, proactive migration can be used to schedule a migration when the state is still small in size. However, both traffic and data patterns might be changing during the deployment of DSP systems, and appropriate and efficient online learning solutions need to be investigated for operator migration.

One promising research direction is to further explore the application of DSP in MEC scenarios. While many previous works have focused on the migration of services, most of them have assumed an all-at-once migration approach. It is worth investigating how partial state migration, state shedding, parallel-track, and distributed checkpoint replication algorithms can be adapted to the more challenging and geo-distributed MEC settings. Specifically, how can these migration mechanisms affect decision-making and change the frequency of migration, compared to the all-at-once approach?

A key challenge in MEC scenarios is the significant increase in migration time due to limited bandwidth and variable hardware resources among potential operator hosts. Therefore, we need to investigate how different migration mechanisms can be applied in such settings, especially in low-bandwidth scenarios where these mechanisms could have the largest benefits. For example, a parallel-track migration mechanism might be a good fit for MEC, as it splits the data streams into one for migration and one for processing. This allows for normal processing to occur while the migration is ongoing, resulting in a smooth handover without discernible downtime. Alternatively, we could consider using the partial state migration approach Megaphone, which could be particularly useful in low-bandwidth scenarios. In contrast, within data centers, these mechanisms may not be necessary, because migration times are significantly lower due to high bandwidth.

VIII. Conclusion

DSP is becoming increasingly important for handling data with high velocity and large variety. The variety is caused by data from different sources and over time as well as other system dynamics, e.g., resource availability, require adapting DSP accordingly. Operator migration is the mechanism for keeping the DSP in an "optimal" configuration over its lifetime. However, operator migration is a complex task that can be solved in many different ways, i.e., there are many design alternatives for operator migration. Which of those alternatives are a good choice depends on factors like the deployment environment, the system goal, workload, etc.

To enable the reader to gain a good understanding of how operator migration works and the design space for it, we introduced a conceptual model of operator migration based on the largest common denominator in the literature to establish a common and unified terminology and taxonomy. In the model, we separated clearly mechanism and policy, i.e., migration mechanism and migration decision. For the latter we placed emphasis on its costs and benefits. The description of existing solutions shall provide the reader with an overview on existing solutions and further foster the understanding of the design alternatives from an algorithmic viewpoint. We complemented this with an empirical study to give the reader some quantitative insights into the impact of different design alternatives for migration mechanisms (i.e., all-at-once and partial state movements), and the impact of the choice of data stream processing system (i.e., Siddhi and Apache Flink). We demonstrate how the freeze time for the naïve all-at-once migration approach is almost 20 times longer than when applying an incremental checkpoint-based partial state migration approach that is based on Rhino [78].

Acknowledgment

The authors thank Fabrice Starks and Stein Kristiansen for insightful discussions on mechanisms of operator migration, and Boris Koldehofe for reviewing and commenting on an earlier version of the manuscript. The authors would like to express their gratitude to the anonymous reviewers for their insightful and constructive feedback.

References

[1] E. Mehmood and T. Anees, "Challenges and solutions for processing real-time big data stream: A systematic literature review," *IEEE Access*, vol. 8, pp. 119123–119143, 2020.

[2] A. Gharaibeh et al., "Smart cities: A survey on data management, security, and enabling technologies," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2456–2501, 4th Quart., 2017.

[3] M. Mohammadi, A. Al-Fuqaha, S. Sorour, and M. Guizani, "Deep learning for IoT big data and streaming analytics: A survey," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 2923–2960, 4th Quart., 2018.

[4] R. Sahal, J. G. Breslin, and M. I. Ali, "Big data and stream processing platforms for industry 4.0 requirements mapping for a predictive maintenance use case," *J. Manuf. Syst.*, vol. 54, pp. 138–151, Jan. 2020.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache fLink: Stream and batch processing in a single engine," *Bull. IEEE Comput. Soc. Techn. Committee Data Eng.*, vol. 36, no. 4, p. 10, 2015.

[6] "fLink." Accessed: Feb. 27, 2023. [Online]. Available: https://flink.apache.org/powered-by

[7] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos. "A survey on the evolution of stream processing systems." 2020. [Online]. Available: https://arxiv.org/abs/2008.00842

[8] J. Manyika et al., *Unlocking the Potential of the Internet of Things*. McKinsey Global Inst., New York, NY, USA 2015.

[9] S. Suhothayan, K. Gajasinghe, I. L. Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, "SIDDHI: A second look at complex event processing architectures," in *Proc. ACM Workshop Gateway Comput. Environ.*, 2011, pp. 43–50.

[10] P. Tucker, K. Tufte, V. Papadimos, and D. Maier, "NexMark—A benchmark for queries over data streams draft," OGI School Sci. Eng., OHSU, Portland, OR, USA, Sep. 2008.

[11] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-based data stream processing," in *Proc. 8th ACM Int. Conf. Distrib. Event Based Syst.*, 2014, pp. 238–245.

[12] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement strategies for Internet-scale data stream systems," *IEEE Internet Comput.*, vol. 12, no. 6, pp. 50–60, Nov./Dec. 2008.

[13] W. Hummer, B. Satzger, and S. Dustdar, "Elastic stream processing in the cloud," *Wiley Interdiscipl. Rev. Data Min. Knowl. Disc.*, vol. 3, no. 5, pp. 333–345, 2013.

[14] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surveys*, vol. 46, no. 4, pp. 1–34, 2014.

[15] M. D. de Assunção, A. da Silva Veith, and R. Buyya, "Distributed data stream processing and edge computing: A survey on resource elasticity and future directions," *J. Netw. Comput. Appl.*, vol. 103, pp. 1–17, Feb. 2018.

[16] Q.-C. To, J. Soto, and V. Markl, "A survey of state management in big data processing systems," *VLDB J.*, vol. 27, no. 6, pp. 847–872, 2018.

[17] H. Röger and R. Mayer, "A comprehensive survey on parallelization and elasticity in stream processing," *ACM Comput. Surveys*, vol. 52, no. 2, pp. 1–37, 2019.

[18] C. Qin, H. Eichelberger, and K. Schmid, "Enactment of adaptation in data stream processing with latency implications—A systematic literature review," *Inf. Softw. Technol.*, vol. 111, pp. 1–21, Jul. 2019.

[19] X. Liu and R. Buyya, "Resource management and scheduling in distributed stream processing systems: A taxonomy, review, and future directions," *ACM Comput. Surveys*, vol. 53, no. 3, pp. 1–41, 2020.

[20] M. Bergui, S. Najah, and N. S. Nikolov, "A survey on bandwidth-aware geo-distributed frameworks for big-data analytics," *J. Big Data*, vol. 8, no. 1, pp. 1–26, 2021.

[21] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Run-time adaptation of data stream processing systems: The state of the art," *ACM Comput. Surveys*, vol. 54, no. 11, pp. 1–36, 2022.

[22] A. Vogel, D. Griebler, M. Danelutto, and L. G. Fernandes, "Self-adaptation on parallel stream processing: A systematic review," *Concurrency Comput. Pract. Exp.*, vol. 34, no. 6, 2022, pp. 1–36, 2022.

[23] T. G. Rodrigues, K. Suto, H. Nishiyama, and N. Kato, "Hybrid method for minimizing service delay in edge cloud computing through VM migration and transmission power control," *IEEE Trans. Comput.*, vol. 66, no. 5, pp. 810–819, Nov. 2017.

[24] O. Osanaiye, S. Chen, Z. Yan, R. Lu, K. R. Choo, and M. Dlodlo, "From cloud to fog computing: A review and a conceptual live VM migration framework," *IEEE Access*, vol. 5, pp. 8284–8300, 2017.

[25] J. Hu, G. Wang, X. Xu, and Y. Lu, "Study on dynamic service migration strategy with energy optimization in mobile edge computing," *Mobile Inf. Syst.*, vol. 2019, Oct. 2019, Art. no. 5794870.

[26] S. K. Pande, S. K. Panda, and S. Das, "Dynamic service migration and resource management for vehicular clouds," *J. Ambient Intell. Humanized Comput.*, vol. 12, pp. 1227–1247, Jan. 2021.

[27] Z. Zeng et al., "Efficient edge service migration in mobile edge computing," in *Proc. IEEE 26th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, 2020, pp. 691–696.

[28] R. Urgaonkar, S. Wang, T. He, M. Zaxfer, K. Chan, and K. K. Leung, "Dynamic service migration and workload scheduling in edge-cloud," *Perform. Eval.*, vol. 91, pp. 205–228, Jan. 2015.

[29] A. Machen, S. Wang, K. K. Leung, B. J. Ko, and T. Salonidis, "Live service migration in mobile edge clouds," *IEEE Wireless Commun.*, vol. 25, no. 1, pp. 140–147, Feb. 2018.

[30] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proc. 2nd ACM/IEEE Symp. Edge Comput. (SEC)*, 2017, pp. 1–13.

[31] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge-clouds," in *Proc. IFIP Netw. Conf. (IFIP Netw.)*, 2015, pp. 1–9.

[32] M. Chen, W. Li, G. Fortino, Y. Hao, L. Hu, and I. Humar, "A dynamic service migration mechanism in edge cognitive computing," *ACM Trans. Internet Technol.*, vol. 19, no. 2, pp. 1–15, Apr. 2019.

[33] S. Wang, R. Urgaonkar, T. He, M. Zafer, K. Chan, and K. K. Leung, "Mobility-induced service migration in mobile micro-clouds," in *Proc. IEEE Mil. Commun. Conf.*, 2014, pp. 835–840.

[34] H. Wang, Y. Li, A. Zhou, Y. Guo, and S. Wang, "Service migration in mobile edge computing: A deep reinforcement learning approach," *Int. J. Commun. Syst.*, vol. 19, no. 2, 2020, Art. no. e4413.

[35] C. Zhang and Z. Zheng, "Task migration for mobile edge computing using deep reinforcement learning," *Future Gener. Comput. Syst.*, vol. 96, pp. 111–118, Jul. 2019.

[36] Z. Gao, Q. Jiao, K. Xiao, Q. Wang, Z. Mo, and Y. Yang, "Deep reinforcement learning based service migration strategy for edge computing," in *Proc. IEEE Int. Conf. Service Oriented Syst. Eng. (SOSE)*, 2019, pp. 116–1165.

[37] L. Ma, S. Yi, N. Carter, and Q. Li, "Efficient live migration of edge services leveraging container layered storage," *IEEE Trans. Mobile Comput.*, vol. 18, no. 9, pp. 2020–2033, Sep. 2019.

[38] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in IoT context: Horizontal and vertical Linux container migration," in *Proc. IEEE Global Internet Things Summit (GIoTS)*, 2017, pp. 1–4.

[39] U. Mandal, M. F. Habib, S. Zhang, B. Mukherjee, and M. Tornatore, "Greening the cloud using renewable-energy-aware service migration," *IEEE Netw.*, vol. 27, no. 6, pp. 36–43, Nov./Dec. 2013.

[40] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1628–1656, 3rd Quart., 2017.

[41] S. Sakr, A. Liu, D. M. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *IEEE Commun. Surveys Tuts.*, vol. 13, no. 3, pp. 311–336, 3rd Quart., 2011.

[42] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource scheduling in edge computing: A survey," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 4, pp. 2131–2165, 1st Quart., 2021.

[43] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 2, pp. 869–904, 2nd Quart., 2020.

[44] S. Yi, C. Li, and Q. Li, "A survey of fog computing: concepts, applications and issues," in *Proc. Workshop Mobile Big Data*, 2015, pp. 37–42.

[45] M. Mukherjee, L. Shu, and D. Wang, "Survey of fog computing: Fundamental, network applications, and research challenges," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 3, pp. 1826–1857, 3rd Quart., 2018.

[46] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "FLUX: An adaptive partitioning operator for continuous query systems," in *Proc. 19th Int. Conf. Data Eng.*, 2003, pp. 25–36.

[47] Y. Xing, S. Zdonik, and J.-H. Hwang, "Dynamic load distribution in the borealis stream processor," in *Proc. IEEE 21st Int. Conf. Data Eng. (ICDE)*, 2005, pp. 791–802.

[48] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik, "A cooperative, self-configuring high-availability solution for stream processing," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, 2007, pp. 176–185.

[49] Y. Zhou, K. Aberer, and K.-L. Tan, "Toward massive query optimization in large-scale distributed stream systems," in *Proc. ACM/IFIP/USENIX Int. Conf. Distrib. Syst. Platforms Open Distrib. Process.*, 2008, pp. 326–345.

[50] W. Hummer, P. Leitner, B. Satzger, and S. Dustdar, "Dynamic migration of processing elements for optimized query execution in event-based systems," in *Proc. OTM Confeder. Int. Conf. Move Meaningful Internet Syst.*, 2011, pp. 451–468.

[51] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD Int. Conf. Manag. Data (SIGMOD)*, ' 2013, pp. 725–736.

[52] C. Lei and E. A. Rundensteiner, "Robust distributed query processing for streaming data," *ACM Trans. Database Syst.*, vol. 39, no. 2, pp. 1–45, 2014.

[53] A. Martin, A. Brito, and C. Fetzer, "Scalable and elastic realtime click stream analysis using Streammine3G," in *Proc. 8th ACM Int. Conf. Distrib. Event Based Syst.*, 2014, pp. 198–205.

[54] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *Proc. IEEE 30th Int. Conf. Data Eng. Workshops*, 2014, pp. 296–302.

[55] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proc. 8th ACM Int. Conf. Distrib. Event Based Syst.*, 2014, pp. 13–22.

[56] E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta, "CAPE: Continuous query engine with heterogeneous-grained adaptivity," in *Proc. 13th Int. Conf. Very Large Data Bases*, vol. 30, 2004, pp. 1353–1356.

[57] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *VLDB J.*, vol. 23, no. 4, pp. 517–539, 2014.

[58] K. G. S. Madsen and Y. Zhou, "Dynamic resource management in a massively parallel stream processing engine," in *Proc. 24th ACM Int. Conf. Inf. Knowl. Manag.*, 2015, pp. 13–22.

[59] A. Martin, T. Smaneoto, T. Dietze, A. Brito, and C. Fetzer, "User-constraint and self-adaptive fault tolerance for event stream processing systems," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2015, pp. 462–473.

[60] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, "Elastic complex event processing exploiting prediction," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, 2015, pp. 213–222.

[61] V. Cardellini, M. Nardelli, and D. Luzi, "Elastic stateful stream processing in storm," in *Proc. Int. Conf. High Perform. Comput. Simulat. (HPCS)*, 2016, pp. 583–590.

[62] K. G. S. Madsen, Y. Zhou, and L. Su, "ENORM: Efficient window-based computation in large-scale distributed stream processing systems," in *Proc. 10th ACM Int. Conf. Distrib. Event Syst.*, 2016, pp. 37–48.

[63] J. Li, C. Pu, Y. Chen, D. Gmach, and D. Milojicic, "Enabling elastic stream processing in shared clusters," in *Proc. IEEE 9th Int. Conf. Cloud Comput. (CLOUD)*, 2016, pp. 108–115.

[64] Y. Liu, X. Shi, and H. Jin, "Runtime-aware adaptive scheduling in stream processing," *Concurrency Comput. Pract. Exp.*, vol. 28, no. 14, pp. 3830–3843, 2016.

[65] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic stream processing for the Internet of Things," in *Proc. IEEE 9th Int. Conf. Cloud Comput. (CLOUD)*, 2016, pp. 100–107.

[66] T. Buddhika, R. Stern, K. Lindburg, K. Ericson, and S. Pallickara, "Online scheduling and interference alleviation for low-latency, high-throughput processing of data streams," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 12, pp. 3553–3569, May 2017.

[67] K. G. S. Madsen, Y. Zhou, and J. Cao, "Integrative dynamic reconfiguration in a parallel stream processing engine," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, 2017, pp. 227–230.

[68] F. Lombardi, L. Aniello, S. Bonomi, and L. Querzoni, "Elastic symbiotic scaling of operators and resources in stream processing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 572–585, Mar. 2018.

[69] C. Wang, X. Meng, Q. Guo, Z. Weng, and C. Yang, "Automating characterization deployment in distributed data stream management systems," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 12, pp. 2669–2681, Dec. 2017.

[70] J. Fang, R. Zhang, T. Z. J. Fu, Z. Zhang, A. Zhou, and J. Zhu, "Parallel stream processing against workload skewness and variance," in *Proc. 26th Int. Symp. High Perform. Parallel Distrib. Comput.*, 2017, pp. 15–26.

[71] L. Mai et al., "CHI: A scalable and programmable control plane for distributed stream processing systems," *Proc. VLDB Endow.*, vol. 11, no. 10, pp. 1303–1316, 2018.

[72] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Optimal operator deployment and replication for elastic distributed data stream processing," *Concurrency Comput. Pract. Exp.*, vol. 30, no. 9, 2018, Art. no. e4334.

[73] J. Fang, R. Zhang, T. Z. J. Fu, Z. Zhang, A. Zhou, and X. Zhou, "Distributed stream rebalance for stateful operator under workload variance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 10, pp. 2223–2240, Oct. 2018.

[74] S. Liu, J. Weng, J. H. Wang, C. An, Y. Zhou, and J. Wang, "An adaptive online scheme for scheduling and resource enforcement in storm," *IEEE/ACM Trans. Netw.*, vol. 27, no. 4, pp. 1373–1386, 2019.

[75] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe, "Megaphone: Latency-conscious state migration for distributed streaming dataflows," *Proc. VLDB Endow.*, vol. 12, no. 9, pp. 1002–1015, 2019.

[76] L. Wang, T. Z. J. Fu, R. T. B. Ma, M. Winslett, and Z. Zhang, "Elasticutor: Rapid elasticity for realtime stateful stream processing," in *Proc. Int. Conf. Manag. Data*, 2019, pp. 573–588

[77] D. Sun, S. Gao, X. Liu, X. You, and R. Buyya, "Dynamic redirection of real-time data streams for elastic stream computing," *Future Gener. Comput. Syst.*, vol. 112, pp. 193–208, Jun. 2020.

[78] B. D. Monte, S. Zeuch, T. Rabl, and V. Markl, "Rhino: Efficient management of very large distributed state for stream processing engines," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2020, pp. 2471–2486.

[79] L. Zhang, W. Zheng, C. Li, Y. Shen, and M. Guo, "AutraScale: An automated and transfer learning solution for streaming system auto-scaling," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2021, pp. 912–921.

[80] R. Gu, H. Yin, W. Zhong, C. Yuan, and Y. Huang, "MECES: Latency-efficient rescaling via prioritized state migration for stateful distributed stream processing systems," in *Proc. USENIX Annu. Techn. Conf. (USENIX ATC)*, 2022, pp. 539–556.

[81] M. K. Geldenhuys, D. Scheinert, O. Kao, and L. Thamsen, "PHOEBE: QoS-aware distributed stream processing through anticipating dynamic workloads," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, 2022, pp. 198–207.

[82] Y. Liu, H. Xu, and W. C. Lau, "Online resource optimization for elastic stream processing with regret guarantee," in *Proc. 51st Int. Conf. Parallel Process.*, 2022, pp. 1–11.

[83] Y. Ahmad, U. Cetintemel, J. Jannotti, A. Zgolinski, and S. B. Zdonik, "Network awareness in Internet-scale stream processing," *IEEE Data Eng. Bull.*, vol. 28, no. 1, pp. 63–69, 2005.

[84] O. Papaemmanouil, U. Cetintemel, and J. Jannotti, "Supporting generic cost models for wide-area stream processing," in *Proc. IEEE 25th Int. Conf. Data Eng.*, 2009, pp. 1084–1095.

[85] T. Repantis and V. Kalogeraki, "Alleviating hot-spots in peer-to-peer stream processing environments," in *Proc. 5th Int. Workshop Databases Inf. Syst. Peer-to-Peer Comput. DBISP2P*, 2007, p. 13.

[86] T. Repantis and V. Kalogeraki, "Hot-spot prediction and alleviation in distributed stream processing applications," in *Proc. IEEE Int. Conf. Depend. Syst. Netw. FTCS DCC (DSN)*, 2008, pp. 346–355.

[87] W. Wang, M. A. Sharaf, S. Guo, and M. Tamer Özsu, "Potential-driven load distribution for distributed data stream processing," in *Proc. 2nd Int. Workshop Scalable Stream Process. Syst.*, 2008, pp. 13–22.

[88] S. Rizou, F. Dürr, and K. Rothermel, "Solving the multi-operator placement problem in large-scale operator networks," in *Proc. 19th Int. Conf. Comput. Commun. Netw.*, 2010, pp. 1–6.

[89] V. Cardellini, F. L. Presti, M. Nardelli, and G. R. Russo, "Decentralized self-adaptation for elastic data stream processing," *Future Gener. Comput. Syst.*, vol. 87, pp. 171–185, Oct. 2018.

[90] T. Hiessl, V. Karagiannis, C. Hochreiner, S. Schulte, and M. Nardelli, "Optimal placement of stream processing operators in the fog," in *Proc. IEEE 3rd Int. Conf. Fog Edge Comput. (ICFEC)*, 2019, pp. 1–10.

[91] H. Röger, S. Bhowmik, and K. Rothermel, "Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures," in *Proc. 20th Int. Middleware Conf.*, 2019, pp. 255–267.

[92] A. Jonathan, A. Chandra, and J. Weissman, "WASP: Wide-area adaptive stream processing," in *Proc. 21st Int. Middleware Conf.*, 2020, pp. 221–235.

[93] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu, "Efficient dynamic operator placement in a locally distributed continuous query system," in *Proc. OTM Conf. Int. Meaningful Internet Syst.*, 2006, pp. 54–71.

[94] G. Brettlecker and H. Schuldt, "Reliable distributed data stream management in mobile environments," *Inf. Syst.*, vol. 36, no. 3, pp. 618–643, 2011.

[95] V. Kakkad, A. E. Santosa, and B. Scholz, "Migrating operator placement for compositional stream graphs," in *Proc. 15th ACM Int. Conf. Model. Anal. Simulat. Wireless Mobile Syst.*, 2012, pp. 125–134.

[96] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran, "MigCEP: Operator migration for mobility driven distributed complex event processing," in *Proc. 7th ACM Int. Conf. Distrib. Event Syst.*, 2013, pp. 183–194.

[97] G. Chatzimilioudis, A. Cuzzocrea, D. Gunopulos, and N. Mamoulis, "A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement," *J. Comput. Syst. Sci.*, vol. 79, no. 3, pp. 349–368, 2013.

[98] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, and U. Ramachandran, "MCEP: A mobility-aware complex event processing system," *ACM Trans. Internet Technol.*, vol. 14, no. 1, pp. 1–24, 2014.

[99] M. Luthra, B. Koldehofe, P. Weisenburger, G. Salvaneschi, and R. Arif, "TCEP: Adapting to dynamic user environments by enabling transitions between operator placement mechanisms," in *Proc. 12th ACM Int. Conf. Distrib. Event Syst.*, 2018, pp. 136–147.

[100] J. Xu and B. Palanisamy, "Model-based reinforcement learning for elastic stream processing in edge computing," in *Proc. IEEE 28th Int. Conf. High Perform. Comput. Data Anal. (HiPC)*, 2021, pp. 292–301.

[101] P. Liu, D. D. Silva, and L. Hu, "DART: A scalable and adaptive edge stream processing engine," in *Proc. USENIX Annu. Tech. Conf.*, 2021, pp. 239–252.

[102] E. Oliveira, A. R. da Rocha, M. Mattoso, and F. C. Delicato, "Latency and energy-awareness in data stream processing for edge based IoT systems," *J. Grid Comput.*, vol. 20, no. 3, p. 27, 2022.

[103] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "StreamCloud: An elastic and scalable data streaming system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2351–2365, Dec. 2012.

[104] B. Lohrmann, P. Janacik, and O. Kao, "Elastic stream processing with latency guarantees," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, 2015, pp. 399–410.

[105] T. De Matteis and G. Mencagli, "Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–12, 2016.

[106] N. Tziritas, T. Loukopoulos, S. U. Khan, C.-Z. Xu, and A. Y. Zomaya, "On improving constrained single and group operator placement using evictions in big data environments," *IEEE Trans. Services Comput.*, vol. 9, no. 5, pp. 818–831, Sep./Oct. 2016.

[107] T. De Matteis and G. Mencagli, "Proactive elasticity and energy awareness in data stream processing," *J. Syst. Softw.*, vol. 127, pp. 302–319, May 2017.

[108] K. Ma, B. Yang, and Z. Yu, "Optimization of stream-based live data migration strategy in the cloud," *Concurrency Comput. Pract. Exp.*, vol. 30, no. 12, 2018, Art. no. e4293.

[109] D. Dedousis, N. Zacheilas, and V. Kalogeraki, "On the fly load balancing to address hot topics in topic-based pub/sub systems," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, 2018, pp. 76–86.

[110] B. Li, Z. Zhang, T. Zheng, Q. Zhong, Q. Huang, and X. Cheng, "Marabunta: Continuous distributed processing of skewed streams," in *Proc. 20th IEEE/ACM Int. Symp. Cluster Cloud Internet Comput. (CCGRID)*, 2020, pp. 252–261.

[111] V. Gulisano, H. Najdataei, Y. Nikolakopoulos, A. V. Papadopoulos, M. Papatriantafilou, and P. Tsigas, "Stretch: Virtual shared-nothing parallelism for scalable and elastic stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 4221–4238, Dec. 2022.

[112] T. Heinze et al., "FUGU: Elastic data stream processing with latency constraints," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 73–81, Jan. 2015.

[113] Y. Wu and K.-L. Tan, "ChronoStream: Elastic stateful stream computation in the cloud," in *Proc. IEEE 31st Int. Conf. Data Eng.*, 2015, pp. 723–734.

[114] L. Xu, B. Peng, and I. Gupta, "STELA: Enabling stream processing systems to scale-in and scale-out on-demand," in *Proc. IEEE Int. Conf. Cloud Eng. (IC2E)*, 2016, pp. 22–31.

[115] X. Ni, S. Schneider, R. Pavuluri, J. Kaus, and K.-L. Wu, "Automating multi-level performance elastic components for IBM streams," in *Proc. 20th Int. Middleware Conf.*, 2019, pp. 163–175.

[116] D. Sun, S. Gao, X. Liu, and R. Buyya, "A multi-level collaborative framework for elastic stream computing systems," *Future Gener. Comput. Syst.*, vol. 128, pp. 117–131, Mar. 2022.

[117] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proc. IEEE 22nd Int. Conf. Data Eng. (ICDE)*, 2006, pp. 49–49.

[118] F. Liu, Z. Jin, W. Mu, W. Zhu, Y. Zhang, and W. Wang, "DROAllocator: A dynamic resource-aware operator allocation framework in distributed streaming processing," in *Proc. Netw. Parallel Comput. 17th IFIP WG 10.3 Int. Conf. (NPC)*, Sep. 2021, pp. 349–360.

[119] "Apache." Accessed: Feb. 26, 2023. [Online]. Available: https://storm.apache.org

[120] "Espertech." Accessed: Feb. 26, 2023. [Online]. Available: https://www.espertech.com/esper

[121] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, "A survey of distributed data stream processing frameworks," *IEEE Access*, vol. 7, pp. 154300–154316, 2019.

[122] "Apache Beam." Accessed: Feb. 26, 2023. [Online]. Available: https://beam.apache.org

[123] E. Volnes, T. Plagemann, V. Goebel, and S. Kristiansen, "EXPOSE: Experimental performance evaluation of stream processing engines made easy," in *Proc. Technol. Conf. Perform. Eval. Benchmarking*, 2020, pp. 18–34.

[124] F. Starks, T. P. Plagemann, and S. Kristiansen, "DCEP-SIM: An open simulation framework for distributed CEP," in *Proc. 11th ACM Int. Conf. Distrib. Event Syst.*, 2017, pp. 180–190.

[125] G. Amarasinghe, M. D. de Assuncao, A. Harwood, and S. Karunasekera, "ECSNet: A simulator for distributed stream processing on edge and cloud environments," *Future Gener. Comput. Syst.*, vol. 111, pp. 401–418, Oct. 2020.

[126] T. Goyal, A. Singh, and A. Agrawal, "CloudSim: Simulator for cloud computing infrastructure and modeling," *Procedia Eng.*, vol. 38, pp. 3566–3572, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877705812023259

[127] H. Gupta, A. V. Dastjerdi, S. K. Ghosh, and R. Buyya, "iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, edge and fog computing environments," *Softw. Pract. Exp.*, vol. 47, no. 9, pp. 1275–1296, 2017.

[128] R. Mahmud, S. Pallewatta, M. Goudarzi, and R. Buyya, "iFogSim2: An extended iFogSim simulator for mobility, clustering, and microservice management in edge and fog computing environments," *J. Syst. Softw.*, vol. 190, Aug. 2022, Art. no. 111351.

[129] C. Sonmez, A. Ozgovde, and C. Ersoy, "EdgeCloudSim: An environment for performance evaluation of edge computing systems," *Trans. Emerg. Telecommun. Technol.*, vol. 29, no. 11, 2018, Art. no. e3493.

[130] T. Qayyum, A. W. Malik, M. A. K. Khattak, O. Khalid, and S. U. Khan, "FogNetSim: A toolkit for modeling and simulation of distributed fog environment," *IEEE Access*, vol. 6, pp. 63570–63583, 2018.

[131] D. N. Jha et al., "IoTSim-edge: A simulation framework for modeling the behavior of Internet of Things and edge computing environments," *Softw. Pract. Exp.*, vol. 50, no. 6, pp. 844–867, 2020.

[132] C. Puliafito et al., "MobFogSim: Simulation of mobility and migration for fog computing," *Simulat. Model. Pract. Theory*, vol. 101, May 2020, Art. no. 102062.

[133] I. Lera, C. Guerrero, and C. Juiz, "YAFs: A simulator for IoT scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91745–91758, 2019.

[134] C. Mechalikh, H. Taktak, and F. Moussa, "PureEdgeSim: A simulation toolkit for performance evaluation of cloud, fog, and pure edge computing environments," in *Proc. Int. Conf. High Perform. Comput. Simulat. (HPCS)*, 2019, pp. 700–707.

[135] M. Salama, Y. Elkhatib, and G. Blair, "IoTNetSim: A modelling and simulation platform for end-to-end IoT services and networking," in *Proc. 12th IEEE/ACM Int. Conf. Utility Cloud Comput.*, 2019, pp. 251–261.

[136] J. Wei, S. Cao, S. Pan, J. Han, L. Yan, and L. Zhang, "SatEdgeSim: A toolkit for modeling and simulation of performance evaluation in satellite edge computing environments," in *Proc. 12th Int. Conf. Commun. Softw. Netw. (ICCSN)*, 2020, pp. 307–313.

[137] K. Alwasel et al., "IoTSim-OSMOSIS: A framework for modeling and simulating IoT applications over an edge-cloud continuum," *J. Syst. Architect.*, vol. 116, Jun. 2021, Art. no. 101956.

[138] V. Cardellini, V. Grassi, F. L. Presti, and M. Nardelli, "Optimal operator replication and placement for distributed stream processing systems," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 44, no. 4, pp. 11–22, 2017.

[139] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz, "Rollback-recovery without checkpoints in distributed event processing systems," in *Proc. 7th ACM Int. Conf. Distrib. Event Syst.*, 2013, pp. 27–38.

[140] "Kafka." Accessed: Jul. 5, 2021. [Online]. Available: https://kafka.apache.org

[141] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman, "Dynamic plan migration for continuous queries over data streams," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2004, pp. 431–442.

[142] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1447–1463, Jun. 2014.

[143] E. Volnes, T. Plagemann, B. Koldehofe, and V. Goebel, "Travel light: State shedding for efficient operator migration," in *Proc. 16th ACM Int. Conf. Distrib. Event Syst.*, 2022, pp. 79–84.

[144] T. N. Pham, N. R. Katsipoulakis, P. K. Chrysanthis, and A. Labrinidis, "Uninterruptible migration of continuous queries without operator state migration," *ACM SIGMOD Rec.*, vol. 46, no. 3, pp. 17–22, 2017.

[145] F. Starks and T. P. Plagemann, "Operator placement for efficient distributed complex event processing in MANETs," in *Proc. IEEE 11th Int. Conf. Wireless Mobile Comput. Netw. Commun. (WiMob)*, 2015, pp. 83–90.

[146] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *Proc. 24th ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst.*, 2005, pp. 250–258.

[147] D. Abadi et al., "The beckman report on database research," *Commun. ACM*, vol. 59, no. 2, pp. 92–99, 2016.

[148] "Gurobi." Accessed: Jan. 23, 2023. [Online]. Available: https://www.gurobi.com

[149] "IBM." Accessed: Jan. 23, 2023. [Online]. Available: http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer

[150] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe, "Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, Oct. 2018, pp. 783–798.

[151] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *J. Syst. Softw.*, vol. 127, pp. 205–216, May 2017.

[152] M. Lindeberg and T. Plagemann, "A study on migration scheduling in distributed stream processing engines," in *Proc. 23rd Int. Conf. Distrib. Comput. Netw.*, 2022, pp. 50–61.

[153] J. F. C. Kingman, "The single server queue in heavy traffic," in *Mathematical Cambridge Philosophical Society*, vol. 57. Cambridge, U.K.: Cambridge Univ. Press, 1961, pp. 902–904.

[154] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer School on Machine Learning*. Berlin, Germany: Springer, 2003, pp. 63–71.

[155] "Weka." Accessed: Aug. 28, 2021. [Online]. Available: https://weka.cms.waikato.ac.nz/

[156] "MOA." Accessed: Aug. 28, 2021. [Online]. Available: https://www.cs.waikato.ac.nz/ml/weka/

[157] S. B. Taieb, G. Bontempi, A. F. Atiya, and A. Sorjamaa, "A review and comparison of strategies for multi-step ahead time series forecasting based on the NN5 forecasting competition," *Exp. Syst. Appl.*, vol. 39, no. 8, pp. 7067–7083, 2012.

[158] "RocksDB." Accessed: Jan. 23, 2023. [Online]. Available: https://rocksdb.org/

[159] M. Abbasi, A. Shahraki, and A. Taherkordi, "Deep learning for network traffic monitoring and analysis (NTMA): A survey," *Comput. Commun.*, vol. 170, pp. 19–41, Mar. 2021.

[160] G. M. Dias, B. Bellalta, and S. Oechsner, "A survey about prediction-based data reduction in wireless sensor networks," *ACM Comput. Surveys*, vol. 49, no. 3, pp. 1–35, 2016.

Espen Volnes received the B.E. and M.E. degrees in computer science from the Department of Informatics, The University of Oslo in 2016 and 2018, respectively, where he is currently pursuing the Ph.D. degree. His research is focused on distributed stream processing systems, with a particular emphasis on operator migration and the performance evaluation, modeling, and simulation of these systems.

Thomas Plagemann received the Dr.Sc. degree in computer science from the Swiss Federal Institute of Technology (ETH), Zürich, Switzerland, in 1994. Since 1996, he has been a Professor with The University of Oslo, Oslo, Norway. He has published over 200 papers in peer-reviewed journals, conferences, and workshops in his field. His research interests include multimodal sensor systems, distributed data stream processing, and mHealth. He received the Medal of ETH Zurich in 1995. For a period of 13 years, he served as the Editor-in-Chief for *Multimedia Systems* (Springer). He is an Associate Editor of the *ACM Transactions on Multimedia Computing, Communication, and Applications*. He is a member of the Association for Computing Machinery.

Vera Goebel received the M.S. degree in computer science from the University of Erlangen–Nuremberg, Erlangen, Germany, in 1989, and the Ph.D. degree in computer science from the University of Zurich, Zürich, Switzerland, in 1994. Since 1997, she has been a Professor with the Department of Informatics, The University of Oslo, Oslo, Norway. She has published over 150 papers in peer-reviewed journals, conferences, and workshops in her field. Her research interests include data management, eHealth, data mining, distributed systems, multimodal sensor systems, and complex event processing.