Datenbank Spektrum (2022) 22:131–141
https://doi.org/10.1007/s13222-022-00415-0

SCHWERPUNKTBEITRAG

NebulaStream: Data Management for the Internet of Things

Steffen Zeuch[1,2] · **Xenofon Chatziliadis**[1] · **Ankit Chaudhary**[1] · **Dimitrios Giouroukis**[1] · **Philipp M. Grulich**[1] · **Dwi Prasetyo Adi Nugroho**[1] · **Ariane Ziehn**[1,2] · **Volker Mark**[1,2]

Received: 4 February 2022 / Accepted: 2 May 2022 / Published online: 30 May 2022

Abstract
The Internet of Things (IoT) presents a novel computing architecture for data management: a distributed, highly dynamic, and heterogeneous environment of massive scale. Applications for the IoT introduce new challenges for integrating the concepts of fog and cloud computing as well as sensor networks in one unified environment. In this paper, we present early approaches that address parts of the overall problem space. All approaches are incorporated into *NebulaStream* (NES), our novel data processing platform that addresses the heterogeneity, unreliability, and scalability challenges of the IoT and thus provides efficient data management for future applications.

Keywords IoT · Streaming · Modern Hardware

1 Introduction

Over the last decade, the amount of produced data has reached unseen magnitudes. The International Data Corporation [1] estimates that by 2025 the global amount of data will reach 175ZB and that 30% of these data will be gathered in real-time. Particularly the number of IoT devices is expected to grow to 20 billion connected devices [2], which are deployed in various application scenarios, e.g. smart-cities [3] and traffic-monitoring [4]. At the same time, devices such as embedded computers or mobile phones continuously increase their processing capabilities. The exploitation of their capabilities becomes essential to handle the future data volumes of the IoT. As a result, the IoT is one of the fastest emerging trends in the area of information and communication technology [5].

The explosion in the number of connected devices triggers the emergence of novel data-driven applications. These applications require low latency, location awareness, geographical distribution, and real-time data processing on millions of data sources. To enable these applications, a data management system needs to leverage the capabilities of IoT devices. To this end, data processing has to expand beyond the cloud [6].

Today's data management systems are not yet ready for these applications. Systems based on the cloud paradigm, e.g., Flink [7], Spark [8], and Kafka Streams [9] do not exploit the capabilities of IoT devices. These system require the centralization of all data in the cloud prior to applying any processing. For future IoT applications, this centralized processing paradigm presents a bottleneck as it requires the collection of data from millions of geo-distributed sensors. This trend will not only impact typical IoT applications like smart cities but also spread across the entire Smart-X universe, e.g., smart city, smart grid, smart home. In addition, cloud-based services and even the HPC community fundamentally neglect that the majority of interesting data is produced outside the cloud [1, 5]. Thus, the main question for future system designs is how to enable analytics on zettabytes of data produced outside the cloud from millions of geo-distributed, heterogeneous devices in real-time.

Compared to the pure cloud-based systems, the IoT introduces many significant changes that require new solutions:

1. **Hierarchical topology:** IoT topologies follow a tree-like structure. Where data moves from the sensors via intermediate nodes to the cloud.
2. **Geo-distribution:** IoT devices expand the cloud and are geo-distributed.

✉ Steffen Zeuch
firstname.lastname@tu-berlin.de

Volker Mark
firstname.lastname@dfki.de

[1] DIMA, TU Berlin, Einsteinufer 17, 10587 Berlin, Germany

[2] IAM, DFKI GmbH, Alt-Moabit 91c, 10559 Berlin, Germany

3. **Heterogeneous devices:** Processing devices range from low-end battery-powered sensors (e.g., Mica Motes) over SoCs (e.g., Raspberry PIs) to high-end servers in the could.
4. **Moving devices:** Devices outside the cloud are potentially movable and change their position within the network topology.
5. **Sensor Management:** Sensor data in IoT environments have a fluctuating nature, which poses a challenge for resource-constrained devices that cannot easily cope with the velocity and volume of incoming data.

To enable future IoT applications and address the introduced changes, a data management system for the IoT has to combine the cloud, the fog, and the sensors in a single unified platform to leverage their unique advantages and enable cross-paradigm optimizations (e.g., fusing, splitting, or operator reordering). A unified environment introduces a previously unprecedented, unique combination of characteristics, e.g., hardware heterogeneity, unreliable nodes, and changing network topologies.

This new set of characteristics enables new cross-paradigm optimizations, which are crucial to support upcoming IoT applications on top of millions of sensors. Overall, there is no general-purpose, end-to-end data management system for a unified sensor-fog-cloud environment with functionality similar to production-ready systems.

In this paper, we present early research results that address parts of the overall problem space. All solutions are designed for *NebulaStream* [1], our novel data processing platform that addresses the heterogeneity, unreliability, and scalability challenges of the IoT to enable efficient data management. We will point out in the respective sections what the current status of integration is. In detail, we present the following approaches: In Sec. 2, we present a new EcoJoin that allows NebulaStream to perform energy-efficient joins on IoT devices. In Sec. 3, we present adaptive query compilation, which allows resource-efficient stream processing on a broad ranges of devices. In Sec. 4, we investigate if state-of-the-art monitoring solutions are suitable for IoT environments with millions of devices and how they should be modified to improve their performance outside the cloud. In Sec. 5, we introduce Governor, an operator placement algorithm designed for the IoT. In Sec. 6, we investigate complex event processing on top of the IoT and propose new solutions for a unified fog-cloud environment that fulfill the low-latency requirements of future IoT applications. In Sec. 7, we investigate adaptive sampling and filtering as important techniques for large-scale sensor deployments. With this paper, we present first research results as results of our work on NebulaStream.

[1] https://nebula.stream

2 Energy Efficiency

IoT systems often offload data processing workload to edge devices due to their proximity to data sources. In many cases, these edge devices operate on a limited power and energy budget, e.g., battery-powered. Hence, it is crucial to take energy efficiency into account when designing a stream processing engine (SPE) for the IoT.

Energy-efficient data management has been an active research area over the few last years. Tsirogiannis et al. [10] argued that energy efficiency can be improved through a more optimized processing efficiency, such as by using a more efficient algorithm. On an orthogonal dimension, *race-to-idle* [11, 12] exploits the capability of modern processors to enter the low-energy idle state when there is no work, e.g., by lowering the clock speed. Another line of research utilize system-on-chip (SoC), such as energy-optimized CPUs [13] and integrated GPUs [14] to deliver high performance and low energy consumption.

Existing research in energy-efficient data management focuses on processing workloads of relational databases. However, the IoT and NebulaStream in particular bring unique challenges that are not tackled by previous work.

2.1 EcoJoin

To tackle the energy efficiency challenges when processing stream join workloads on IoT devices we proposed the *EcoJoin* [15] for NebulaStream. In particular, EcoJoin addresses three aspects of IoT devices when handling energy efficiency. First, we exploit changing workload characteristics. Second, we design a join algorithm to increase high computational efficiency. Third, we utilize the available hardware to preserve energy utilization.

2.1.1 Exploiting Workload Characteristics.

EcoJoin adopts the idle-to-race approach to achieve energy-efficient stream join processing. To exploit idle-to-race, EcoJoin splits the processing of incoming event streams into batches. In particular, EcoJoin applies a two-step approach: first, by setting the processor to enter an idle phase when waiting for tuples in batches and second, by adjusting the processor's frequency to the value required to sustain the current ingestion rate.

2.1.2 Increasing Computational Efficiency.

EcoJoin applies a symmetric hash join algorithm to achieve higher throughput and energy efficiency compared to the nested loop join algorithm that is used in state-of-the-art stream join solutions. Adapting the symmetric hash join to a streaming setup leads to two challenges. First, triggering the build and probe for each incoming tuple is inefficient. Second, a full scan of the hash table to invalidate a tuple that exceeds the range of the specified window is expensive. To this end, EcoJoin implements a batching mechanism and reduces the tuple invalidation overhead by only evicting records if the hash table is close to its maximum capacity.

Under the hood, EcoJoin maintains two hash tables, i.e., one for each join side, and performs the join operation in three phases. In the first phase, EcoJoin inserts tuples from a batch that is ready processing into the corresponding hash table, which is partitioned on the tuple's key. In the second phase, EcoJoin probes the batch from one side with the hash table of the other join side. If a pair satisfy the join predicate, EcoJoin emits the matching pair into a result stream. In the third phase, EcoJoin deletes all invalid tuples, i.e., tuples that are outside of the current window range and thus will not find any match. To reduce the clean-up overhead, EcoJoin counts invalid tuples and only triggers the clean-up process periodically.

2.1.3 Exploiting Heterogeneous Processors.

EcoJoin leverages CPUs and integrated GPUs to maximize energy efficiency. To this end, EcoJoin switches devices to process the join based on their availability and under the given workload.

2.2 EcoJoin Evaluation

We evaluated the power consumption, throughput, and latency of EcoJoin. To this end, we compared the throughput and energy consumption of EcoJoin to existing stream join algorithms, i.e., naive Handshake Join (CPU) [16] and HELLS Join (CPU/GPU co-processing) [17]. Figure 1 shows the throughput and power consumption of the different stream join algorithms. The x-axis shows the maximum sustainable throughput of each algorithm. The CPU and GPU variant of our EcoJoin achieve the highest throughput compared to other algorithms, i.e., up to 1M tuples/s. In contrast, the Handshake Join and HELLSJoin achieve similar maximum sustainable throughput of 4K tuples/s,

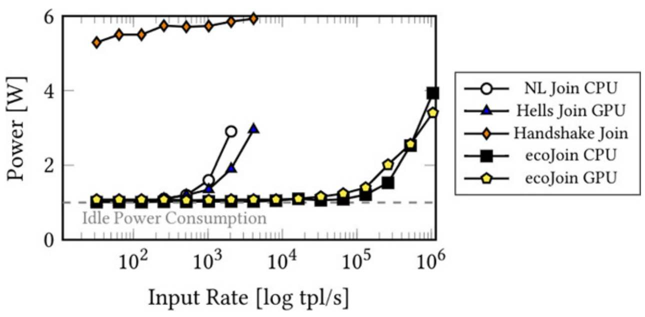

Fig. 1 Power consumption of streaming joins

while the naive NLJ achieves only 2K tuples/s. On the y-axis, we show the amount of power consumed by each join algorithm. EcoJoin consumes significantly less energy compared to other join algorithms for the same sustainable throughput. In particular, EcoJoin shows up to 81% less power consumption compared to Handshake Join and 65% less compared to HELLS Join. In summary, the EcoJoin outperforms state-of-the-art stream join algorithms in terms of throughput and energy efficiency.

2.3 Summary

Energy efficiency is one of the most crucial aspects in offloading data processing workloads to edge devices in an IoT data management system. We tackle this challenge within EcoJoin by proposing an energy-efficient stream join. In future work, we plan to integrate the approach that we have in EcoJoin to NES and investigate the feasibility of a similar approach for other stream processing operators in general.

3 Efficient Stream Processing

Over the last years, the increasing demands of real-time use-cases has led to a wide adoption of stream processing workloads. These workloads execute long-running queries over unbounded, continuously changing, high-velocity data streams. State-of-the-art stream processing engines, e.g., Flink [7], and Storm [18], distribute processing to large homogeneous compute clusters to achieve high-throughput and low-latency's. However, these systems are designed for cloud environments and can not address the unique requirements of IoT scenarios. In particular, we observed in an experimental study three fundamental limitations of current SPEs [19].

3.1 Low resource utilization.

Our study revealed that none of the existing systems were able to fully utilize the available hardware resources. In particular, they suffered from instruction cache misses as they follow interpretation-based processing model and data cache misses as they rely on managed runtimes. Consequently, these systems are not suited for IoT environments, which require efficient data processing on a wide range of heterogeneous and resource-constrained devices.

3.2 Inefficient parallilization.

State-of-the-art SPEs utilize a *key-by* partitioning to distribute stream processing workloads uniformly across large compute clusters. However, this strategy introduces an expensive shuffle phase, which induces a high overhead per node. Consequently, this strategy is not suited for low-end and mid-end IoT devices with limited processing power.

3.3 Missed optimizations.

Current SPEs receive a query, apply optimizations, and deploy an execution plan. However, stream processing queries are inherently long-running. Thus, the data characteristic of the input stream may change over the run time, which reduces the efficiency of a plan. As a result, current SPEs are not designed to handle the dynamicity of the IoT.

To address these three limitations, NebulaStream leverages adaptive query compilation [20].

3.4 Adaptive Query Compilation

In the following, we discuss NebulaStream's adaptive query compilation engine, which bases on Grizzly [20]. Grizzly combines query compilation, task-based parallelization, and adaptive optimizations to increase resource utilization.

Query compilation is a well-known technique for efficient data processing in relational data processing engines [21, 22]. Grizzly adopts this approach for stream processing and supports the unique properties of stream processing queries. Within queries, Grizzly fuses operators to compact code fragments and performs all operations in a single pass over the data. Furthermore, Grizzly avoids serialization and accesses data directly. As a result, query compilation based on Grizzly increases code and data locality within NES.

Task-based parallelization enables the concurrent execution of operator pipelines [23, 24] to fully utilize multicore CPUs. This eliminates the overhead of data pre-partitioning but requires coordination between threads. To this end, Grizzly introduces a lock-free window operator.

Adaptive optimizations enable a system to react to changing data characteristics [25, 26]. Grizzly monitors data-characteristics, detects changes, and generates new code variants at run time. This allows Grizzly to perform speculative optimizations that exploit assumptions about the incoming data. To mitigate profiling overhead, Grizzly leverages hardware-performance counters to detect changes in the data characteristics.

By leveraging these techniques within NebulaStream, we improve execution performance significantly. Our results show that Grizzly outperforms state-of-the-art SPEs by up to an order of magnitude without losing generality.

3.5 Summary

NebulaStream applies adaptive query compilation to mitigate the limitations of state-of-the-art SPEs. As a result, NebulaStream is able to fully utilize the available hardware resources. In the future, we plan to extend NebulaStream with support of UDFs [27] and concept drift detection [28].

4 Monitoring

SPEs apply optimizations to enable efficient data processing at scale, e.g., state management [29], operator placement [30–34], scheduling [35, 36], query compilation [20], adaptive sampling [37], and load shedding [38]. For optimal decision making these approaches require accurate system metrics of the underlying infrastructure as well as applications metrics of the running tasks. To provide these metrics, SPEs need to monitor the infrastructure (e.g., available resources on the devices, utilized bandwidth), detects node failures, and measures the performance of internal workloads (e.g., operator throughput).

Managing the variety of metrics at scale in a highly distributed cloud-based SPE is in general a challenging task for a monitoring system. Monitoring an SPE in an IoT environment exacerbates the challenge even further due to the geo-distributed, heterogeneous, highly dynamic, and volatile nature of IoT environments [39]. The collected metrics can become quickly outdated due to device or network failures and fluctuating load. While there exist several cloud-based performance monitoring solution [40–42], there are no solutions for IoT systems like NebulaStream.

In previous work [43], we analyzed two common approaches for performance monitoring in cloud-based SPEs and investigate their applicability in large-scale IoT settings. The first approach uses an external, general-purpose monitoring system to monitor the performance of the SPE. In contrast, the second approach implements monitoring internally within the SPE. As the first approach is widely adopted in industry, we experimentally evaluate whether it can be applied efficiently in an IoT setting. Finally, based on our analysis, we highlight the need to re-design monitoring frameworks for IoT data management systems and sketch a set of requirements.

4.1 State of the Art

External monitoring systems such as Ganglia [40], Nagios [41], JCatascopia [44], the Elastic ecosystem [45] or Prometheus [42] consist of four major components: monitoring agents, monitoring server, data storage analytics & visualization. Such frameworks monitor SPEs as follows: 1) the monitoring agents collects metrics from the nodes of an SPE, 2) send the metrics to the monitoring server for processing, and 3) transmit the processed metrics back to the master node of the SPE. This solution has two major drawbacks. On the one hand, it creates a strong dependency between the SPE and an external system, which makes the dependent components harder to maintain in case of changes. On the other hand, metrics have to cross multiple system boundaries, i.e., from the SPE to the monitoring system and back to the SPE, which creates an unnecessary overhead [19].

To alleviate the inefficiencies of external monitoring systems, some cloud-based SPEs like Flink, Spark, and Storm implement their own monitoring internally. From a high-level architectural perspective the monitoring of cloud-based SPEs consists of the following common components: metrics manager, SPE components, master node, and external monitoring system. The metrics manager retrieves performance metrics from the Java Virtual Machine (JVM) instance and from internal components of the SPE. Afterwards, the metrics are forwarded to the corresponding destinations, which can be components inside the workers that require monitoring data, the master node, or external systems. By introducing such a coupling of the monitoring component, the SPE can avoid the additional overhead of external monitoring systems and make monitoring more efficient.

4.2 Requirements

Existing general purpose monitoring solutions can be seamlessly implemented and integrated in cloud-based SPEs that use JVMs. There are many Java libraries that facilitate the gathering of system and user metrics independently from the underlying operating system and hardware. Data management systems for the IoT like NES are, however, implemented in C++ due to its efficiency and suitability for low-end devices [19]. Additionally, IoT environments are more complex and diverse than the purely cloud-based environments, since they frequently undergo changes and consist of many hierarchical levels with different networks and permissions. As a result, not all devices can be connected directly at all times.

In [43], we propose a list of requirements that are specifically tailored to an internal monitoring component for a data management system for the IoT. Specifically, we identify four categories of functional requirements that such a system needs to satisfy in order to enable monitoring in IoT environments: 1) performance optimization and scalability, 2) handling uncertainties, 3) permission and access control and 4) handling heterogeneity. The first category addresses the resource constrained environment and massive number of nodes properties. The second category addresses the dynamic topology property. The third one handles the complex networks property, and the last category addresses the properties diversity.

4.3 Summary

In this section we explored existing monitoring solutions with respect to their applicability in a stream processing setting for IoT environments. We described the architecture of SPE-external monitoring frameworks and provided an overview of SPE-internal monitoring components. We conclude that novel monitoring solutions are required to support the distribution, heterogeneity, volatility, and complexity of IoT environments.

5 Operator Placement

An IoT infrastructure encompasses a large number of heterogeneous, geo-distributed, and sparsely connected devices [39]. These devices are capable of producing large volumes of data that allow interesting real-time applications such as public mobility, health care, or manufacturing [6]. These applications require IoT workloads which can process massive amounts of real-time data to generate new insights. The state-of-the-art solutions first collect these large volumes of data centrally in the cloud and then employ cloud-based frameworks such as, Flink or Spark, for data processing. However, this central collection of data leads to an increase in processing latency and overall network, storage, and compute resources.

The resources available at and close to IoT devices in the fog [46] infrastructure can mitigate these challenges by performing in-network processing, such as partial-aggregation or filtering. This pre-processing allows reduction in the overall data and further processing requirements. However, holistic computation of data is not possible within the fog infrastructure due to unavailability of data from IoT devices in an isolated network zone.

A unified fog-cloud infrastructure allows mitigating the challenges of increased resource consumption at cloud infrastructure and inability of performing holistic computations at fog infrastructure. In particular, by unifying the cloud and fog infrastructure we can enable real-time processing of data and efficient utilization of underlying heterogeneous and geo-distributed resources.

An IoT workload can have different application objectives, such as infrastructure monitoring, time-sensitive anomaly detection, or fault-tolerant billing information collection. The heterogeneity and volatility in the network and compute resources in this unified fog-cloud infrastructure presents unique challenges for achieving these application objectives. A common challenge in such unified infrastructure is to perform placement of operators from a workload to leverage the unique infrastructure properties and the workload Service Level Objectives (SLOs), e.g., high-throughput, low resource consumption.

Table 1 Five example Governor policies

	Low-Latency	Fault-Tolerance	High-Throughput	Minimum Resource Consumption	Minimum Energy Consumption
Path Selection Phase	– Distinct paths with lowlink latency	– All paths between sources and sink	– Distinct paths with high bandwidth capacity	– Common path between sources and sink	– Common path between sources and sink
Operator Assignment Phase	– Non-blocking operators closer to source – Replicate operators when possible	– Use shared nodes among selected paths – Replicate operators when possible	– Non-blocking operators closer to source – Blocking operators closer to sink	– Share intermediate operators among different sources – Avoid operator replication	– Non-blocking operators closer to source – Share intermediate operators among different sources – Avoid operator replication

The cloud-centric operator placement techniques ignores volatility and heterogeneity in the infrastructure and instead focus on network and compute resource efficiency [47–49]. In contrast, approaches proposed for unified fog-cloud environments consider volatility and heterogeneity but only optimize for a specific goal, e.g., network efficiency or fault tolerance [30, 50, 51]. None of the above works allow specification of custom SLOs for operator placement.

We proposed *Governor* [31], a novel operator placement approach for the placement of workloads with varying SLO in a unified fog-cloud environment. Governor consists of *Governor Policies* (GPs) and the Governor placement process which are discussed in remainder of this section.

5.1 Governor Polices

Governor policies consist of a sub-set of heuristic-based rules from a fixed collection of rules. An administrator prepares the GP by selecting rules from a predefined catalog, which is maintained and updated by a domain expert. A GP then guides the operator placement process for a specific SLO. For example, to place a query with high fault-tolerance as an SLO, a GP selects all available network paths between source and sink nodes, and places replicated operators on different nodes along the selected paths. In case of an operator failure, another replica operator will take over the failed operator's workload, thereby achieving the fault-tolerance objective.

A GP defines rules for two purpose: *path selection* and *operator assignment*. The rules defined for the path selection guide the selection of a subset of available paths in the infrastructure for data transfer. In contrast, the rules defined for operator assignment guide the placement of operators on nodes on the selected paths. Table 1 presents five different example policies with different SLO objectives. We can notice that different policies share common heuristic rules among each other.

5.2 Governor Placement Process

The Governor's operator placement process consists of four different steps. Figure 2 presents steps involved in the placement of operators of a query on an example infrastructure. First, Governor transforms a query into a directed acyclic graph (DAG) of interconnected operators (Fig. 2a). Second, Governor extracts a graph representing underlying physical

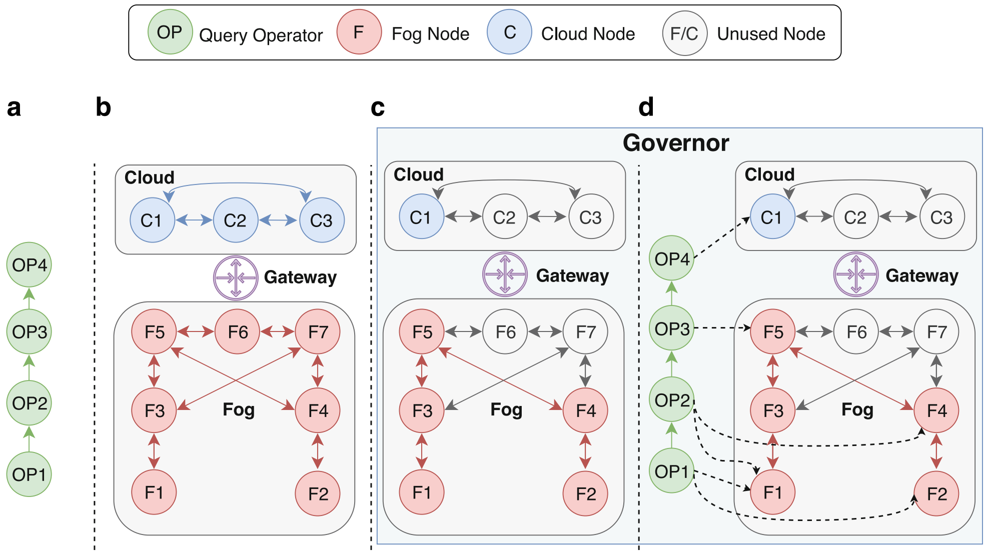

Fig. 2 Governor placement process

infrastructure containing compute nodes that are interconnected by network links (see Fig. 2b). In the next two steps, Governor uses the GP to guide the placement process to assign the operators from the query DAG on the infrastructure. Third, Governor uses the rules from a GP to perform the path selection from the graph representing the infrastructure (Fig. 2c). Governor identifies paths between the source (IoT device) and the sink (cloud servers) nodes using the heuristic rules defined in the GP. The path selection phase allows pruning the search space for the operator assignment in a large-scale infrastructure containing thousands of interconnected nodes. Fourth, Governor performs operator assignment on nodes along the selected paths from the previous step (Fig. 2d). Governor distinguishes between operators residing on a fixed location (i.e., *pinned*) and the operators that can be placed freely on any node along the path (i.e., *unpinned*).

5.3 Summary

In this section, we presented Governor, a first step towards operator placement on a unified fog-cloud infrastructure. We have implemented few example Governor policies and its overall operator placement strategy in NebulaStream. We presented Governor's ability to accept custom Policies with different optimization objectives for operator placement. With that, we introduced the challenges as well as early solutions for operator placement in the future IoT era.

6 Complex Event Processing

Complex event processing (CEP) is a stream processing method that detects user-defined patterns in data streams. Patterns contain information about the relationships between event types, e.g., by time or cause [52]. When matches of the pattern are detected in the involved streams, the user is notified, e.g., by pushing a notification, or the application may react autonomously, e.g., showing a popup on a live map. As a result, CEP is a powerful and necessary tool for monitoring IoT applications, e.g., traffic congestion monitoring, smart street lamps [53], and an essential feature in NebulaStream [39] we currently investigate.

6.1 State-of-the-Art

In order to enable CEP for the IoT, we need engines that allow for large-scale and massive distributed processing. Thus, we focus our state-of-the-art analysis on CEP solutions that allow for efficient stream processing in cloud or fog environments. Giatrakos et al. [54] and Carbone et al. [55] reviewed a broad set of parallel and distributed CEP approaches and concluded that no single solution provides the entire set of features that are required to leverage the cloud entirely, e.g., flexible resource allocation [56] or multi-query optimization [57]. In contrast, over the last two decades, analytical stream processing (ASP) evolved for Big Data processing with engines that scale out over clusters of machines to cope with the volume and velocity of data, e.g., Flink [7] and Spark [8]. This trend also adapts original CEP features, e.g., out-of-order arrivals and event time [55, 58]. Therefore, first CEP approaches have been proposed that leverage cloud-optimized ASP engines to provide large-scale CEP features. One naive approach is to run instances of CEP engines on worker nodes of an ASP engine-managed cluster. This approach was demonstrated by *'Siddhi on Spark'* [59] but is conceptually not limited to this combination. However, this setup still prevents the full leverage of the cloud. Another approach enables built-in support for CEP in an ASP engine to utilize its distributed stream processing optimization. Flink is the only ASP that provides this approach with the FlinkCEP library [60].

6.2 Solution Sketch

We aim to leverage NebulaStreams's unified fog-cloud environment for CEP and provide a solution that fulfills the latency requirements of IoT applications. To this end, we investigate in the following core features of CEP.

6.2.1 Pattern Evaluation Mechanisms

Efficient CEP requires a high-performance pattern evaluation mechanism. FlinkCEP uses an order-based evaluation mechanism, i.e., an NFA applicable to NebulaStream's cloud layer. However, these solutions introduce two problems: First, cloud bottlenecks, e.g., the central data collection, remain without leveraging the fog layer. Second, cloud CEP solutions are naive and bound to one common evaluation mechanism, i.e., order-based evaluation driven by automata. The structure of automata is order-based and allows for limited pattern plan optimization, while other evaluation mechanisms, e.g., tree-based solutions, have an enormous optimization potential similar to join query plan optimization [61] (see Fig. 3). In order to overcome these problems, we are currently evaluating promising evaluation mechanisms for distributed pattern detection and bringing them together with the fog paradigm and distribution strategies. To this end, we consider three candidate solutions: (1) the order-based mechanism used by FlinkCEP, (2) the tree-based mechanism that provides more extensive pattern plan optimization techniques as order-based mechanisms, and (3) a mapping between CEP operators and ASP operators that leverages cloud-optimized operator implementations.

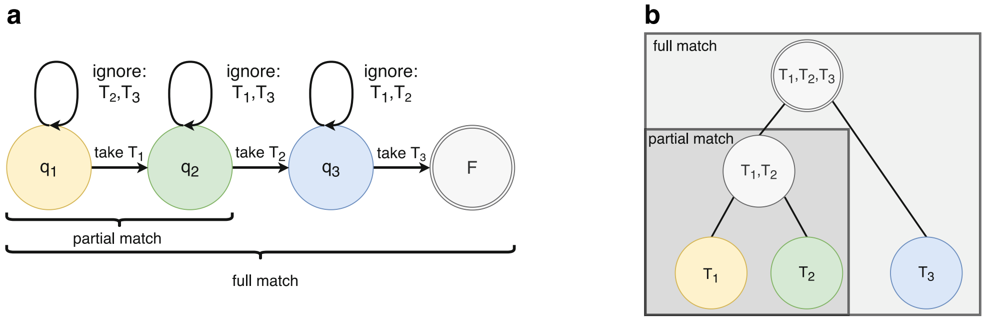

Fig. 3 Pattern Detection Mechanism. **a** order-based (NFA), **b** tree-based

6.2.2 Optimization of Evaluation Mechanisms

After identifying the best candidate for an evaluation mechanism in fog-cloud environments, we suggest focusing on its optimization for the fog environment. Unlike the cloud paradigm, fog environments allow us to tailor the data to the relevant only using the fog nodes on the data flow through the network. Cloud-based optimization techniques do not consider the limitations and challenges of using fog nodes, i.e., unreliable and moving low-end devices in a dynamic network topology. These limitations require adjustments to stream processing in general, e.g., derive query plans for dynamically changing instead of static networks. Another challenge specific to CEP is that CEP is a stateful processing method that requires storing partial matches (intermediate results) in such a dynamic environment. Storing and maintaining large amounts of partial matches is already a significant challenge in cloud environments with almost unlimited resources. We currently investigate two CEP optimization techniques that need to be adjusted for fog execution: rewriting single patterns and sharing techniques for multi patterns [62]. Both techniques improve pattern detection plans by reducing partial matches.

6.3 Summary

We introduced the importance of CEP for the IoT and its state-of-the-art solutions. Furthermore, we discussed current challenges and sketched directions to enable CEP for NebulaStream's fog-cloud environment [63]. We are currently working on implementing the above-introduced pattern evaluation mechanisms for NebulaStream that will form a baseline for further optimizations to leverage the fog layer.

7 Sensor Data Management

The Internet of Things (IoT) creates environments with millions of heterogeneous sensor nodes that provide data in real time [64]. Timely acquisition of data from such a highly distributed sensor deployment poses complex challenges for data management systems. The main research challenges of these sensors networks are: (i) heterogeneity of resources, (ii) widely distributed communication networks, and (iii) nodes with diverse sets of capabilities [39, 65, 66].

In NES, we examine how current research tackles those challenges in highly distributed sensor environments. To this end, we want to first collect classes of algorithms that enable data management in very large IoT environments and consuquently implement and expand the State-of-the-Art inside NES, as none of the catalogued techniques have been evaluated in the context of real-time stream processing.

In previous work [37], we consider *adaptive sampling* and *filtering algorithms* that run on the data sources, i.e., directly on sensor nodes, as the fundamental building blocks for a sensor-aware system. With adaptive sampling and filtering, we are able to address important scalability challenges for gathering volatile streams of heterogeneous sensor data from the IoT. Adaptive sampling enables the system to decide *when*, *where*, and *how often* to sample [67] a value from a sensor. In contrast, adaptive filtering allows the system to decide *which* values to transmit to an SPE for further analysis. To highlight the unique characteristics of adaptive sampling and filtering, we summarize briefly both techniques.

7.1 Adaptive Sampling

Adaptive sampling changes the sampling rates on a sensor node such that (i) sensors observing an interesting event provide detailed data (high sampling rate) and (ii) sensors that do not observe interesting events reduce sampling rates to not overload the receiver. Ideally, at any time, a subset of sensors dynamically switches to a higher sampling rate while the majority of sensors provides data at lower rates. As a result, a highly adaptive sampling approach is crucial to enable future IoT deployments with millions of sensor nodes.

In Fig. 4, blue diamonds mark sensor readings performed with a fixed rate. In contrast, orange triangles mark sensor readings performed with an adaptive rate. In this example, adaptive sampling has three advantages compared to fixed rate sampling. First, it performs fewer sensor reads, which

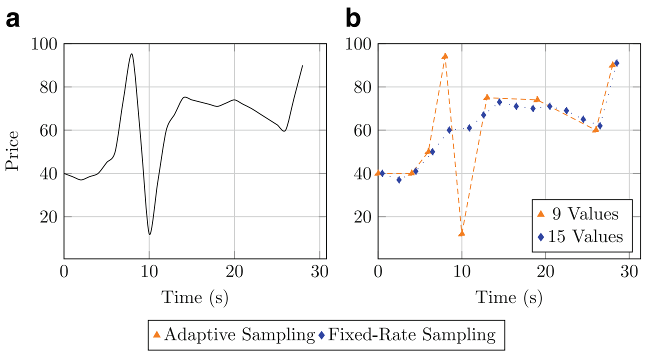

Fig. 4 Adaptive Sampling. **a** original stream, **b** resulting stream

leads to energy savings on the sensor nodes. Second, it transmits fewer values to the central analysis engine, which saves network traffic. Third, the reconstructed phenomenon of the adaptive technique is closer to the original.

7.2 Adaptive Filtering

Adaptive filtering techniques focus on finding a threshold that helps deciding whether a system should transmit a sensor value. In particular, if a sensor value is similar to previous values or evolves predictably, a node can avoid data transmission and thus save network traffic. In contrast, if a value changes unexpectedly and does not follow a prediction, a node needs to transmit an update to maintain the precision of the reconstructed signal on the receiver side. Since the behavior of a signal may change frequently, static a-priori filtering would lead to sub-optimal decisions. Thus, filter thresholds and rules for value filtering must adapt over time to the observed values.

Figure 5 depicts an example of adaptive filtering. This example compares two different filtering techniques (left and right). Both techniques transmit the values for the intervals marked in red for a stream of price changes. However, on the left, the technique transmits changes that exceed the predefined threshold only, ignoring later changes. In contrast, the right side captures changes depending on the magnitude of change between values. A technique with fixed thresholds may underestimate the importance of changes. By adapting the threshold, a filtering technique reduces network traffic and maintains data quality.

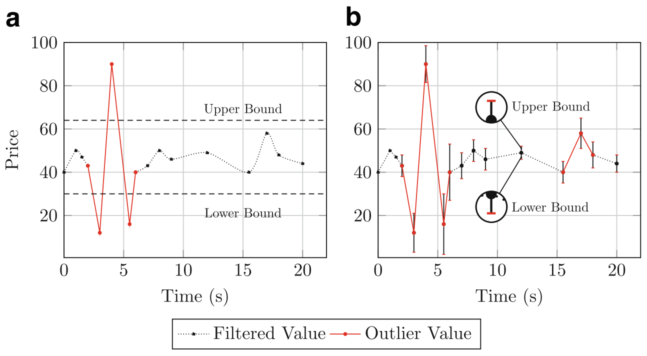

Fig. 5 Adaptive Filtering. **a** static filter, **b** adaptive filter

7.3 Summary

We discussed fundamental techniques that will enable NebulaStream to process data from millions of devices, in real time. Adaptive sampling from sensors and adaptive filtering of sampled data allow NebulaStream to get accurate information while reducing the amount of network messages to a bare minimum. Currently, we are working on the implementation of equivalent operators inside NES in order to create a true sensor data management system for the scale of the IoT.

8 Conclusion

In this paper, we presented early research results in the huge problem space of IoT environments. We summarized our approaches and outlined why they are important to create an IoT data management system. All approaches are incorporated into NebulaStream, our novel data processing platform that addresses the heterogeneity, unreliability, and scalability challenges of the IoT and thus provides efficient data management for future applications. Besides the discussed areas, we investigate in NebulaStream different directions of development, e.g., modern networking infrastructure [68], the support of machine learning [69, 70], and the optimization of stream operators [71–74].

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Reinsel D et al (2018) Data age 2025: The digitization of the world from edge to core. https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf. Zugegriffen: 15. Dez. 2019

2. Hung M (2017) Leading the iot, gartner insights on how to lead in a connected world (Gartner Research)
3. Gavriilidis H et al (2020) Scaling a public transport monitoring system to internet of things infrastructures. In: EDBT
4. Grulich PM, Zukunft O (2017) Bringing big data into the car: Does it scale? In: Innovate-Data, IEEE
5. Miorandi D, Sicari S, De Pellegrini F, Chlamtac I (2012) Internet of things: Vision, applications and research challenges. Ad Hoc Netw 10:1497–1516. https://doi.org/10.1016/j.adhoc.2012.02.016
6. Zeuch S et al (2020) Nebulastream: Complex analytics beyond the cloud. In: VLIoT
7. Alexandrov A et al (2014) The stratosphere platform for big data analytics. VLDB J 23:939–964. https://doi.org/10.1007/s00778-014-0357-y
8. Zaharia M et al (2016) Apache spark: a unified engine for big data processing. Commun ACM 59:56–65. https://doi.org/10.1145/2934664
9. Sax MJ et al (2018) Streams and tables: Two sides of the same coin. In: BIRTE
10. Tsirogiannis D et al (2010) Analyzing the energy efficiency of a database server. In: SIGMOD, ACM
11. Götz S et al (2014) Energy-efficient databases using sweet spot frequencies. In: UCC, IEEE
12. Kissinger T et al (2018) Adaptive energy-control for in-memory database systems. In: SIGMOD, ACM
13. Ungethüm A et al (2015) Query processing on low-energy many-core processors. In: ICDE, IEEE
14. Cheng X et al (2015) Energy-efficient query processing on embedded CPU-GPU architectures. In: DaMoN, ACM
15. Michalke A et al (2021) An energy-efficient stream join for the internet of things. In: DaMoN, ACM
16. Teubner J, Müller R (2011) How soccer players would do stream joins. In: SIGMOD, ACM
17. Karnagel T et al (2013) The hells-join: a heterogeneous stream join for extremely large windows. In: DaMoN, ACM
18. Kulkarni S et al (2015) Twitter heron: Stream processing at scale. In: ACM SIGMOD
19. Zeuch S et al (2019) Analyzing efficient stream processing on modern hardware. PVLDB 12:516–530. https://doi.org/10.14778/3303753.3303758
20. Grulich PM et al (2020) Grizzly: Efficient stream processing through adaptive query compilation. In: ACM SIGMOD
21. Breß S et al (2018) Generating custom code for efficient query execution on heterogeneous processors. VLDB J 27:797–822. https://doi.org/10.1007/s00778-018-0512-y
22. Neumann T (2011) Efficiently compiling efficient query plans for modern hardware. VLDB 4:539–550. https://doi.org/10.14778/2002938.2002940
23. Leis V et al (2014) Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In: ACM SIGMOD
24. Zeuch S, Freytag J (2014) QTM: modelling query execution with tasks. In: ADMS
25. Răducanu B et al (2013) Micro adaptivity in vectorwise. In: SIGMOD, ACM
26. Zeuch S et al (2016) Non-invasive progressive optimization for in-memory databases. Proc VLDB Endow 9:1659–1670. https://doi.org/10.14778/3007328.3007332
27. Grulich PM et al (2021) Babelfish: Efficient execution of polyglot queries. Proc VLDB Endow 15:196–210. https://doi.org/10.14778/3489496.3489501
28. Grulich PM et al (2018) Scalable detection of concept drifts on data streams with parallel adaptive windowing. In: EDBT
29. Del Monte B et al (2020) Rhino: Efficient management of very large distributed state for stream processing engines. In: SIGMOD
30. Cardellini V et al (2016) Optimal operator placement for distributed stream processing applications. In: ACM DEBS
31. Chaudhary A et al (2020) Governor: Operator placement for a unified fog-cloud environment. In: EDBT
32. Pietzuch P et al (2006) Network-aware operator placement for stream-processing systems. In: ICDE
33. Rizou S et al (2010) Solving the multi-operator placement problem in large-scale operator networks. In: ICCCN
34. Xu J et al (2014) T-storm: Traffic-aware online scheduling in storm. In: IEEE ICDCS
35. Babcock B et al (2003) Chain: Operator scheduling for memory minimization in data stream systems. In: ACM SIGMOD
36. Carney D et al (2003) Operator scheduling in a data stream manager. In: VLDB
37. Giouroukis D et al (2020) A survey of adaptive sampling and filtering algorithms for the internet of things. In: DEBS
38. Babcock B et al (2004) Load shedding for aggregation queries over data streams. In: IEEE ICDE
39. Zeuch S et al (2020) The nebulastream platform for data and application management in the internet of things. In: CIDR
40. Massie ML et al (2004) The ganglia distributed monitoring system: design, implementation, and experience. Parallel Comput 30:817–840. https://doi.org/10.1016/j.parco.2004.04.001
41. Nagios Enterprises Nagios xi 5.6.10. https://www.nagios.com. Zugegriffen: 6. Mai 2021
42. Prometheus Authors Prometheus 2.15.9. https://prometheus.io/. Zugegriffen: 6. Mai 2021
43. Chatziliadis X et al (2021) Monitoring of stream processing engines beyond the cloud: An overview. OJIOT 7:71–82
44. Trihinas D et al (2014) Jcatascopia: Monitoring elastically adaptive applications in the cloud. In: CCGrid
45. Elastic Elastic ecosystem. https://www.elastic.co. Zugegriffen: 14. Juni 2021
46. Bonomi F et al (2012) Fog computing and its role in the internet of things. In: MCC
47. Chatzistergiou A et al (2014) Fast heuristics for near-optimal task allocation in data stream processing over clusters. In: CIKM, ACM
48. Huang Y et al (2011) Operator placement with qos constraints for distributed stream processing. In: CNSM, IEEE
49. Kafil M et al (1998) Optimal task assignment in heterogeneous distributed computing systems. Ieee Concurr 6:42–50. https://doi.org/10.1109/4434.708255
50. Cardellini V et al (2017) Optimal operator replication and placement for distributed stream processing systems. SIGMETRICS 44:11–22. https://doi.org/10.1145/3092819.3092823
51. da Silva Veith A et al (2018) Latency-aware placement of data stream analytics on edge computing. In: International conference on service-oriented computing. Springer, Heidelberg
52. Luckham DC (2005) The power of events - an introduction to complex event processing in distributed enterprise systems. ACM, New York
53. Ahmed A et al (2019) Fog computing applications: Taxonomy and requirements. CoRR
54. Giatrakos N et al (2020) Complex event recognition in the big data era: a survey. VLDB J 29:313–352. https://doi.org/10.1007/s00778-019-00557-w
55. Carbone P et al (2017) Large-scale data stream processing systems. In: Handbook of big data technologies
56. Mei Y, Madden S (2009) Zstream: a cost-based query processor for adaptively detecting composite events. In: SIGMOD, ACM
57. Cugola G et al (2012) Complex event processing with T-REX. J Syst Softw 85:1709–1728. https://doi.org/10.1016/j.jss.2012.03.056
58. Luckham D (2019) What's the difference between esp and cep? https://complexevents.com/2019/07/15/whats-the-difference-between-esp-and-cep-2/. Zugegriffen: 06.2020
59. Stratio decision. https://github.com/Stratio/Decision. Zugegriffen: 01.2021

60. Flinkcep (2019) Complex event processing for flink. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html. Zugegriffen: 12.2019
61. Kolchinsky I, Schuster A (2018) Join query optimization techniques for complex event processing applications. Proc Vldb Endow 11:1332–1345. https://doi.org/10.14778/3236187.3236189
62. Kolchinsky I, Schuster A (2019) Real-time multi-pattern detection over event streams. In: SIGMOD, ACM
63. Ziehn A (2020) Complex event processing for the internet of things. In: VLDB PhD Workshop
64. Traub J et al (2017) Optimized on-demand data streaming from sensor nodes. In: SoCC
65. Gaura EI et al (2013) Edge mining the internet of things. IEEE Internet Things J 8:10220–10221. https://doi.org/10.1109/JIOT.2021.3075304
66. Yao Y et al (2015) Edal: An energy-efficient, delay-aware, and lifetime-balancing data collection protocol for heterogeneous wireless sensor networks. TON
67. Madden SR et al (2005) Tinydb: an acquisitional query processing system for sensor networks. TODS 30:122–173. https://doi.org/10.1145/1061318.1061322
68. Del Monte B et al (2022) Rethinking stateful stream processing with rdma. In: SIGMOD
69. Grulich PM, Nawab F (2018) Collaborative edge and cloud neural networks for real-time video processing. In: VLDB
70. Baunsgaard S et al (2021) ExDRa: Exploratory data science on federated raw data. In: SIGMOD
71. Benson L et al (2020) Disco: Efficient distributed window aggregation. In: EDBT
72. Traub J et al (2018) Scotty: Efficient window aggregation for out-of-order stream processing. In: ICDE, IEEE
73. Traub J et al (2019) Efficient window aggregation with general stream slicing. In: EDBT
74. Traub J et al (2021) Scotty: General and efficient open-source window aggregation for stream processing systems. TODS