Open Access

Open Journal of Internet of Things (OJIOT)
Volume 7, Issue 1, 2021
http://www.ronpub.com/ojiot
ISSN 2364-7108

Monitoring of Stream Processing Engines Beyond the Cloud: An Overview

Xenofon Chatziliadis[A] Eleni Tzirita Zacharatou[A] Steffen Zeuch[A,B] Volker Markl[A,B]

[A]Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany
[B]DFKI GmbH, Trippstadter Str. 122, 67663 Kaiserslautern, Germany
x.chatziliadis@tu-berlin.de, eleni.tziritazacharatou@tu-berlin.de, steffen.zeuch@dfki.de, volker.markl@dima.tu-berlin.de

Abstract

The Internet of Things (IoT) is rapidly growing into a network of billions of interconnected physical devices that constantly stream data. To enable data-driven IoT applications, data management systems like NebulaStream have emerged that manage and process data streams, potentially in combination with data at rest, in a heterogeneous distributed environment of cloud and edge devices. To perform internal optimizations, an IoT data management system requires a monitoring component that collects system metrics of the underlying infrastructure and application metrics of the running processing tasks. In this paper, we explore the applicability of existing cloud-based monitoring solutions for stream processing engines in an IoT environment. To this end, we provide an overview of commonly used approaches, discuss their design, and outline their suitability for the IoT. Furthermore, we experimentally evaluate different monitoring scenarios in an IoT environment and highlight bottlenecks and inefficiencies of existing approaches. Based on our study, we show the need for novel monitoring solutions for the IoT and define a set of requirements.

Type of Paper and Keywords

Research Paper: *stream processing, performance monitoring, IoT data management*

1 Introduction

Stream processing engines (SPEs) such as Spark [30], Flink [1], Storm [17], and Kafka Streams [16] are nowadays widely used for various applications that require managing and processing data in real-time. Some example applications are location-tracking services, fabrication line management, and network management [24]. SPEs are designed to distribute the workload horizontally across thousands of servers that are usually located in data centers on-premise or hosted by various cloud providers. However, the upcoming Internet of Things (IoT) triggered an ongoing effort to develop an IoT data management system that can exploit the processing capabilities of cloud-external devices in addition to the cloud servers. One recently proposed data management system for the IoT is NebulaStream (NES) [31, 33]. NES manages and processes data streams, potentially in combination with data at rest, in a heterogeneous distributed environment of cloud and (potentially mobile) edge devices. In contrast to prior engines, NES can cope with the heterogeneity and distribution of compute and data, deal with potentially unreliable communication, and constantly evolve under continuous operation. Overall, NES extends the processing capabilities of data management systems beyond the cloud, which is becoming increasingly important as the amount of edge devices is rising.

This paper is accepted at the *International Workshop on Very Large Internet of Things (VLIoT 2021)* in conjunction with the VLDB 2021 conference in Copenhagen, Denmark. The proceedings of VLIoT@VLDB 2021 are published in the Open Journal of Internet of Things (OJIOT) as special issue.

To enable efficient data processing at scale, SPEs apply several system-internal optimizations. To this end, there are various approaches that address different performance problems, e.g., state management [9], operator placement [6, 8, 21, 23, 29], scheduling [3, 7], query compilation [14], adaptive sampling [12], and load shedding [4]. To perform optimization decisions, all 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 a monitoring component that monitors the infrastructure (e.g., available resources on the devices, utilized bandwidth), detects node failures, and measures the performance of internal components (e.g., operator throughput). In addition to being performant and scalable, the monitoring component should be robust, non-intrusive, interoperable across different infrastructures, and should support live migration [25].

Monitoring the performance of a highly distributed cloud-based SPE is already challenging. Monitoring an SPE in an IoT environment exacerbates the challenge even further, as IoT environments are geo-distributed, very heterogeneous, highly dynamic, and volatile [31]. The collected metrics can become quickly outdated due to device or network failures and fluctuating load. In particular, cloud-based systems run on a well-defined infrastructure of high-end servers with reliable network connections and scale up to thousands of nodes. In contrast, IoT data management systems have to scale to millions or even billions of nodes and incorporate edge devices that use limited network bandwidth and are prone to failures and disconnections [20]. Furthermore, future IoT-driven applications have to process thousands of user queries that are running in parallel [31], which makes the system load very ad-hoc and unpredictable. Finally, while there exist several industry-established performance monitoring solutions for cloud-based systems [18, 19, 22], there are no established solutions for IoT data management systems like NES, as these systems represent an emerging technology field.

In this paper, we analyze 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, despite being designed for the cloud. 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.

In the remainder of the paper, we first explore commonly used SPEs with respect to the architecture of their monitoring components and discuss their integration with external general-purpose monitoring systems like Prometheus (cf. Section 2). After that, we evaluate experimentally the applicability of a general-purpose monitoring solution (i.e., Prometheus) for monitoring a cluster of typical IoT devices in Section 3. Finally, we highlight the need for novel monitoring solutions and specify a set of requirements for efficient performance monitoring for data management systems in the IoT in Section 4, before concluding in Section 5.

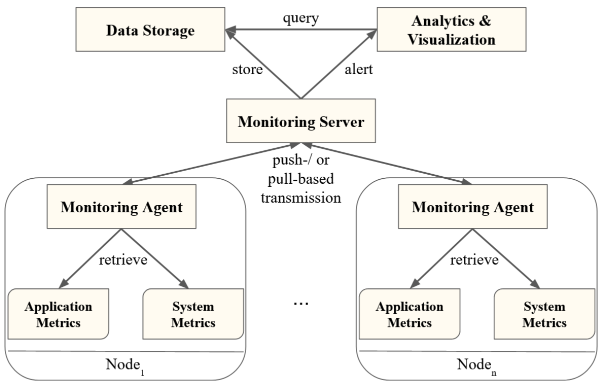


Figure 1: Architecture of external monitoring systems

2 Monitoring of Stream Processing Engines in the Cloud

Stream processing engines like Flink require two types of metrics for internal decisions [5, 34]. First, they require system metrics of the underlying infrastructure (e.g., containers, virtual machines, or bare-metal processes) like available memory, bandwidth, or CPU utilization. Second, they require application metrics that are generated by internal components of the SPE (e.g. throughput of different operators). System and application metrics can be obtained either externally using third-party, general-purpose monitoring systems or internally using a build-in component within the SPE. Next, we review state-of-the-art solutions for external monitoring in Section 2.1 and internal monitoring in Section 2.2, before drawing general conclusions in Section 2.3.

2.1 SPE-External Monitoring

External monitoring systems like Ganglia [18], Nagios [19], JCatascopia [27], the Elastic ecosystem [11] or Prometheus [22] consist of four major components (c.f., Figure 1): 1) monitoring agents, 2) monitoring server, 3) data storage, 4) analytics & visualization. The monitoring agents are daemons running on different nodes of the compute topology. Nodes are usually servers, virtual machines, or containers. The monitoring agents are responsible to retrieve performance metrics of nodes (e.g., CPU utilization or memory consumption) as well as application-specific metrics. The agents transmit then these metrics to the monitoring server via push- or pull-based techniques. With pull-based techniques, the metrics are collected on demand

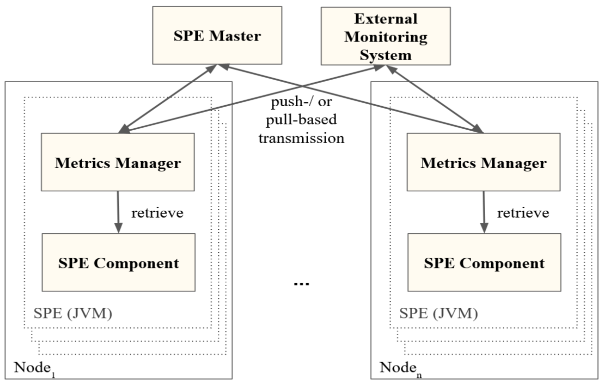


Figure 2: Monitoring in cloud-based SPEs

by using, e.g., a RESTful API. In contrast, push-based techniques collect metrics continuously by forwarding them based on given conditions, e.g., at fixed time intervals. Depending on the topology, some monitoring systems also allow to organize monitoring agents hierarchically, such that metrics can be aggregated in intermediate layers using either push- or pull-based techniques to reduce the overall amount of data. The monitoring server receives the metrics from the agents and processes them through grouping, aggregation, or by creating alerts or notifications based on events or thresholds [18, 19]. Monitoring servers can be deployed on either physical or virtual instances and do not need to reside on the same node as the monitoring agents. The data storage persists the collected metrics. Since metrics contain both a value and a timestamp that indicates the time of the measurement, they are typically stored in key-value stores like InfluxDB [15]. After the metrics have been processed and archived, they are presented to users together with alerts and notifications via graphs or dashboards in the analytics & visualization component. Common tools for analytics and visualization are Grafana [13] or Kibana [11].

External monitoring frameworks monitor SPEs as follows: 1) the monitoring agents collect metrics from the worker nodes and the master of the 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 then again back to the SPE, which creates an unnecessary overhead [32]. Monitoring an SPE using solely external monitoring systems is thus inefficient. To alleviate this inefficiency, some cloud-based SPEs like Flink, Spark, and Storm implement their own internal monitoring components, which we discuss in the next section.

2.2 SPE-Internal Monitoring

From a high-level architectural perspective, cloud-based SPEs (e.g., Flink, Spark, and Storm) consist of the following common components for monitoring: 1) metrics manager, 2) SPE components, 3) master node, and 4) external monitoring system (cf. Figure 2). 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.

In the remainder of this section, we focus on three important aspects of performance monitoring in SPEs. Section 2.2.1 describes the gathering of metrics, i.e., the process of collecting metrics from monitored components. Section 2.2.2 describes the management of metrics, i.e., how the SPE administrates and represents metrics internally. Finally, Section 2.2.3 illustrates the integration of the SPE with external third-party monitoring systems and discusses how the SPE makes metrics available to these systems.

2.2.1 Metric Gathering

Common cloud-based SPEs have an integrated component for gathering system and application metrics [1, 17, 30]. Flink provides a pull-based and a push-based approach for collecting metrics. In the pull-based approach, all worker nodes expose an endpoint, e.g., REST or RPC. The destination node is then able to query the metrics from all nodes whenever new or updated values are required. In contrast, the push-based approach is initiated from the worker nodes. Hereby, a configuration on each node defines when the transmission of metrics has to be triggered. Triggers are commonly time-based (e.g. transmit data every 10s) or event-based (transmit data after a given threshold has been exceeded). Flink and Spark enable the retrieval of monitoring data through a metrics API. In Storm/Heron, the monitoring component is called metrics manager. As Flink, Spark, and Storm/Heron run within a JVM, their monitoring components retrieve system metrics through Java libraries like the Dropwizard Metrics API [10].

Figure 3 illustrates the performance monitoring in Flink. Each monitored component contains an instance of a metric query service and a metric reporter. The metric query service is used for the pull-based approach, while the metric reporter exposes metrics in a push-based manner. In the pull-based approach, the metrics fetcher periodically queries and aggregates monitoring data from all components via the metric query service. The aggregated data is visualized in a web-fronted in the form of dashboards, thereby helping users understand and debug the framework. The communication regarding fetching metrics is asynchronous, which can result

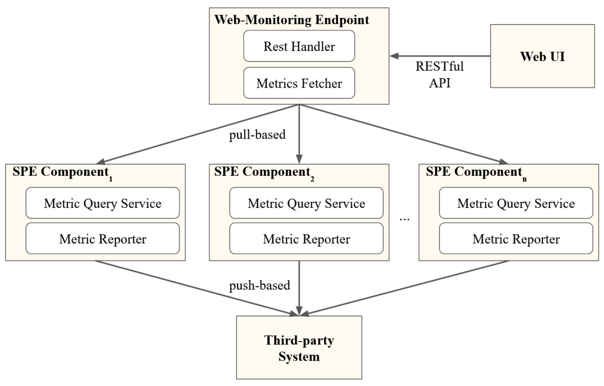


Figure 3: Performance monitoring in Apache Flink [2]

in timeouts during the fetching process and outdated data.

In the push-based approach, the metric reporters collect metrics from each node and transmit them without aggregation to a specified destination such as an external monitoring system or a database.

2.2.2 Metric Management

For metric management, Flink supports four different metric types (based on the Dropwizards library [10]): Gauge, Counter, Histogram, and Meter. Spark additionally supports a Timer type. These metric types are specified as follows:

- **Gauge** measures a value that can arbitrarily increase or decrease, like CPU utilization.
- **Counter** is a Gauge representing a long value that is updated atomically. Counters can be incremented and decremented.
- **Histogram** measures the statistical distribution of values in a stream of data such as minimum, maximum, mean, median, standard deviation, and different percentiles.
- **Meter** measures the number of events in a unit of time, e.g., requests per second. Statistics measured using meters are, for example, mean rate or 15-minute moving averages. In the context of an SPE, a meter can be described as a sliding window with a given aggregation function.
- **Timer** measures duration. Spark, for example, uses this metric type to measure the processing time of messages.

Metrics are assigned to metric groups, i.e., named metric containers. Metric groups are combined to create a nested hierarchy based on group names. Each metric group can be uniquely identified by its name and place in the hierarchy. Metric groups enable the reporting and gathering of metrics at different granularities, making metrics management intuitive. Each group can contain different types of metrics, e.g., Gauges, Counters, Histograms, Meters, or Timers.

Table 1: Excerpt of a metric group hierarchy in Flink

Scope	Infix	Metric	Type
Job-/ TaskManager	Status. JVM.Memory	Heap.Used	Gauge
		Heap.Committed	Gauge
		Heap.Max	Gauge
Task	Shuffle. Netty.Input	NumBytesInLocal	Counter
		NumBytesInLocalPerSec	Meter

In Flink, metrics belong conceptually to two different groups which are specified by an attribute called scope. For each metric, there exist a user-defined scope and a system-provided scope. The user-defined scope is optional and enables custom grouping. In contrast, the system-defined scope is the top-level group in the hierarchy and contains context information about the metric, e.g., in which task it was registered or to what job that task belongs to. Table 1 shows an excerpt of two metric groups. The upper one contains three metrics about the heap memory of the JVM, which are all of type Gauge. The group at the bottom contains two metrics regarding the network communication via Netty [26]. $NumBytesInLocal$ describes the total number of bytes a task has read from a local source. $NumBytesInLocalPerSec$ represents the number of bytes that a task is reading from a local source per second.

2.2.3 Integration with External Systems

In a cloud environment, there is the need to collect various performance metrics from different running frameworks such as the SPE, as well as from the operating system and the storage layer. One arising requirement is thus having a unified monitoring solution that can both ensure the correctness of all running processes in the SPE and enable the monitoring of the infrastructure. To that end, it is required to extend the metric stack of the SPE to support custom code and instrumentation. Flink realizes this with custom reporters and rich functions, while Spark has custom listeners and plugins. Flink provides reporters for sending metrics to common systems such as InfluxDB and Prometheus. In addition, users can implement custom reporting via inheriting the metrics reporter interface. Rich functions are extensions of user defined functions (UDFs) that grant access to runtime information and state variables. In Spark, the listener interface enables accessing the runtime by intercepting events from the Spark scheduler. Plugins are external packages that register additional custom-defined metrics and are executed at the startup of the executors and the driver.

2.3 Summary

In summary, cloud-based monitoring solutions are mature and widely adopted in industry. External monitoring systems collect system and application metrics via monitoring agents, which are independent processes running on every node. Stream processing engines implement monitoring internally to avoid crossing of system boundaries and enable customizable gathering of internal performance metrics. Despite the fact that SPEs and monitoring systems differ architecture- and functionality-wise, they rely on the same key concepts to retrieve and transmit metrics. More precisely, both systems retrieve metrics via push- or pull-based approaches and transmit metrics based on given conditions which are usually static time intervals, thresholds, or events. Furthermore, they both aggregate metrics at the server or master node. The monitoring approaches that are employed by state-of-the-art SPEs are designed for cloud infrastructures. However, IoT environments are significantly different. They are geo-distributed, have a much larger scale, and involve volatile low-end devices. To examine the suitability of current solutions in these emerging environments, we conducted an experimental evaluation that we present in the next section.

3 Experimental Evaluation

Cloud-based monitoring is mature and widely adopted in industry. In this section, we evaluate experimentally whether it can also be applied efficiently in an IoT setting, despite being designed for the cloud. To this end, we examine the impact of the sampling period and the number of transmitted metrics on CPU usage, memory utilization, and network traffic. For our experiments, we chose the Prometheus ecosystem as a representative solution, since it is widely adopted in practice, simple to use, and well-documented (see Section 3.1.2 for more details). The goal of our experiments is to identify the current limitations that have to be addressed to enable efficient monitoring for novel data management systems for the IoT. To simulate an IoT topology, we use a set of four Raspberry Pis (RPIs). We use one RPI as a server for aggregating the monitoring data, and the remaining three RPIs as monitored nodes. In the remaining section, we first describe the setup of our experiments in Section 3.1, before presenting our experimental findings in Section 3.2.

3.1 Experimental Setup

Next, we first specify the hardware of the RPIs in Section 3.1.1 and then the software that we are running on them in Section 3.1.2. Finally, we describe the experimental design in Section 3.1.3.

3.1.1 Hardware

We use the RPI 4 Model B to create a small-scale, low-end IoT cluster, i.e., a cluster with low CPU, memory, and network bandwidth capacities. Each of our RPIs is equipped with a 1.5 GHz 64-bit quad core ARM Cortex-A72 processor, on-board 802.11ac Wi-Fi, Bluetooth 5, full gigabit Ethernet and 4GB of LPDDR4-3200 SDRAM. The RPIs are connected over an Ethernet switch and communicate via statically assigned IP addresses. For the RPI's energy supply, we use a central USB power hub.

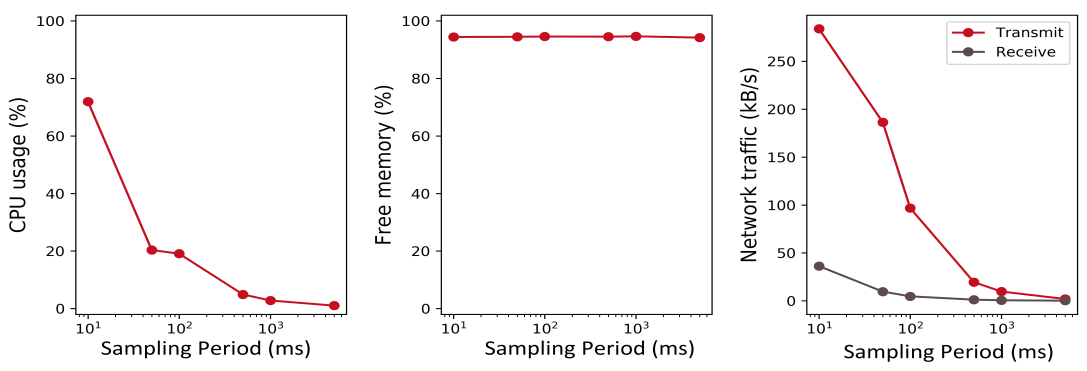
3.1.2 Software

There is a large variety of available monitoring solutions [18, 19, 22, 27]. For our experiments, we decided to use Prometheus [22] as a monitoring system together with the dashboard framework Grafana for the visualization. Prometheus is broadly adopted in the industry and thus has a large community, extensive documentation, and a rich ecosystem of exporters that enable monitoring data from various sources. For our experiments, we use the *Node Exporter*, which gathers hardware metrics provided by the operating system. The Node Exporter runs on all monitored RPIs and exposes an HTTP endpoint that is queried by Prometheus. Furthermore, each exporter consists of a set of collectors that can be enabled on demand to provide the respective monitoring information (e.g., a CPU or file system collector). Grafana is a similarly well-established dashboard framework that has many pre-configured dashboards to display hardware monitoring data from Prometheus. We use Grafana to simulate the additional load of an end-user dashboard solution on both Prometheus and the RPI, but we exclude the resource consumption of Grafana from our measurements. To investigate to what extent an edge device is capable of acting as a local monitoring aggregator, we use one of the RPIs as the server for Prometheus and Grafana

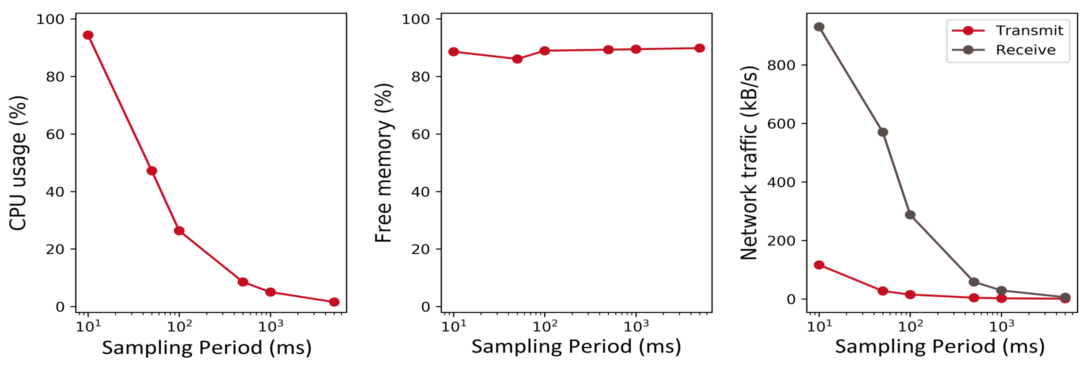
3.1.3 Experimental Design

The goal of our experiments is to evaluate the impact of different monitoring scenarios on the RPI cluster. Specifically, we investigate the impact of the sampling period at which new samples are queried from Node Exporters and of the number of monitored metrics. We use an idle Node Exporter process that has a network connection to the server node as a baseline. The monitoring overhead is then calculated by measuring the difference in the resource consumption with respect to this baseline.

Metrics: In our experiments, we evaluate two different categories of metrics. The first category contains standard measurements for the CPU, memory, and network bandwidth usage. We do not measure IO usage since the Node Exporter does not use the file system. The second category consists of two Prometheus-specific metrics. The *scrape duration* describes the latency of a request that Prometheus sends to the Node Exporter, i.e., the time that it takes to receive a new value from a node at the server. The *failure rate* specifies the fraction of scrape jobs that fail due to request timeouts or other errors that occur when the devices are under load.



(a) Monitored Node



(b) Server Node

Figure 4: CPU, memory and network bandwidth consumption over different sampling periods (number of monitored metrics = 591). Other free parameters are kept fixed at their default values.

Measurements: We perform our measurements over a five-minute time frame with a rate of 1 measurement/second. We retrieved the hardware measurements using *nmon* and the Prometheus-specific ones by querying the Prometheus database. Since our measurements showed only little variance over the five-minute time frame, we only show a single measurement in each of our result figures.

3.2 Experimental Results

This section presents the results of two experiments: one where we vary the sampling period, and one where we vary the number of monitored metrics.

3.2.1 Sampling Period Experiments

In the first part of this experiment shown in Figure 4, we varied the sampling period (x-axis) and analyzed the effect on three metrics (y-axis): average CPU usage (left), [1] free memory (middle), and network traffic (right). The top of the figure shows the results for the server node and the bottom shows the average results over all monitored nodes. Regarding CPU usage, the results show an almost exponential increase for both the server and monitored nodes as the sampling period decreases. When looking at the absolute values, we see a CPU consumption of more than 20 percent for sampling periods less than 500 ms. Considering that in this experiment the monitoring system is the sole resource consumer, i.e., there is no other running computation, we have revealed that on low-end devices such our RPIs, a sampling period smaller than 500 ms leaves few resources for the actual processing. The memory consumption does not vary significantly with the sampling period and does not differ significantly from the baseline. For the network consumption, we see a linear increase with decreasing sampling period for the transmitting channel on the monitored node and the receiving channel on the server node, respectively. This is expected, as with a smaller sampling period, more metrics are generated, which need to be transmitted to the server node. [2]

In the second part of this experiment shown in Figure 5,

[1] We additionally investigated the CPU usage per core and confirmed an even distribution of CPU load before considering the overall CPU usage.

[2] Note that the linear increase is not depicted clearly in Figure 4 due to the non-linear scale of the x-axis.

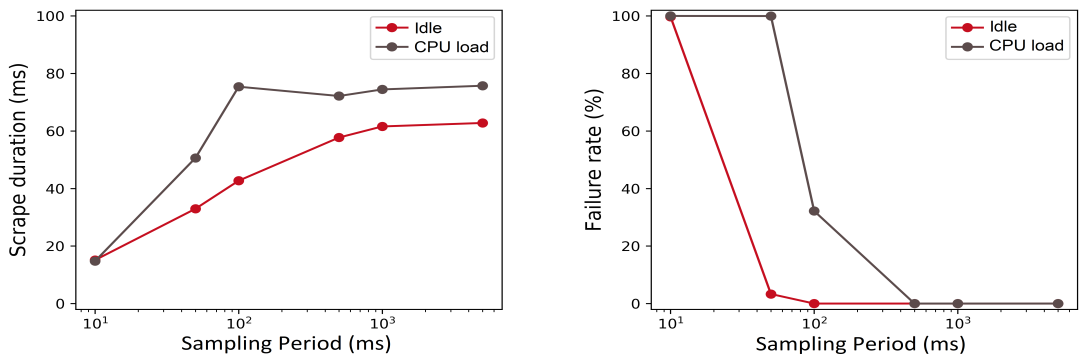


Figure 5: Scrape duration and failure rate of scrape jobs over different sampling periods

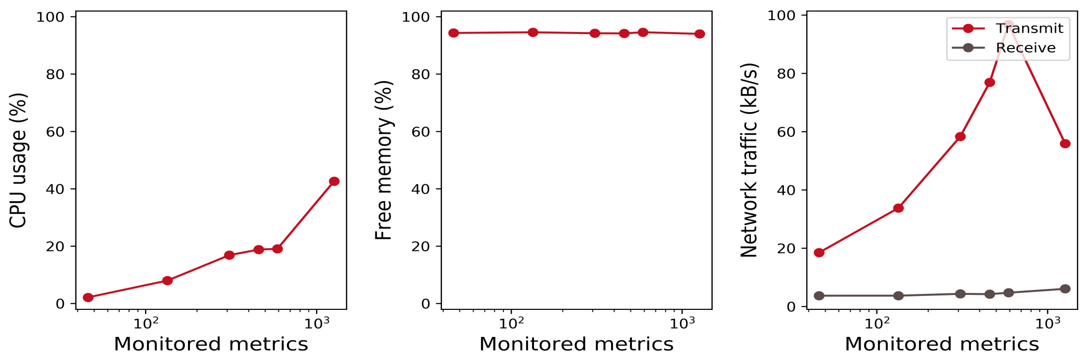


Figure 6: CPU, memory and network bandwidth consumption on a monitored node over different numbers of monitored metrics (sampling period = 100 ms)

we varied the sampling period (x-axis) and measured the corresponding scrape duration (left) and failure rate (right), which are displayed on the y-axis. In addition to the scenario in which only Node Exporter is running on a monitored node (red curve), we also measured the scrape duration and the failure rate when the monitored node is under full CPU load (black curve). The load has been artificially created via *stress*. The results show a generally increasing scrape duration with larger sampling periods. Full CPU load generally increases the scrape duration by about 15 to 20 milliseconds. This effect occurs due to the decreased CPU resources that the monitored node devotes to the Node Exporter. At ca. 100 ms the scrape duration for both the idle and full CPU load setup starts decreasing, because the scrape jobs are starting to fail. The scrape duration for both setups meets at 10 ms due to the fact that almost 100% of the scrape jobs are failing at that point. In these cases, the Node Exporter is unable to process metrics within the requested sampling period, and a connection timeout occurs. Consequently, no data is sent back in the HTTP response making the processing faster. Looking at the failure rates, we see that high CPU load causes a large number of scrape jobs to fail for low sampling periods. Since real-world applications are likely to impose a considerable CPU load on the nodes, we validate our previous conclusion that a sampling period of less than 500 ms is not feasible in such a setup.

In summary, the sampling period experiments have shown that Prometheus cannot handle small sampling periods efficiently. We identified that the CPU usage increases exponentially and the network traffic linearly with decreasing sampling periods. For an IoT topology that usually consists of millions of low-end devices with limited resources and bandwidth, this showcases that current solutions are not applicable. To avoid a system overload, a monitoring framework for the IoT should support efficient management of varying sampling periods. Additionally, we identified that Prometheus is not capable to collect and transmit performance metrics in a reliable way when the device has limited resources and is under heavy CPU load. In all experiments that were performed in these conditions, we were experiencing timeouts and lost packets. Such behaviour occurs less frequently on cloud servers with many resources and fast network connections, but becomes especially problematic for an IoT data management systems like NES that aims to offload work to edge devices.

3.2.2 Number of Metrics Experiment

In this experiment shown in Figure 6, we varied the number of monitored metrics (x-axis) and analyzed the effect on three metrics (y-axis): average CPU usage (left), free memory (middle), and network traffic (right). The available interval steps for the number of exposed metrics are implicitly defined by the number of metrics that each single collector exposes. By default, Node Exporter exposes 591 metrics in our setup. After measuring the number of metrics per collector, we use a series of collectors that in total expose 46, 134, 309, 459, 591, and 1268 metrics. The large gap between the two last steps is caused by the $systemd$ collector that exposes more than 600 metrics by itself. For this experiment, we fix the sampling period to 100 ms. We chose a small sampling period to emphasize the differences caused by the number of metrics.

Analyzing the results, we see that the CPU usage increases linearly with the number of monitored metrics. We observe a slightly higher increase in CPU usage in the last measuring point, which occurs due to the additional overhead the *systemd* collector is generating. All in all, the CPU usage is, however, behaving as expected, since the system has to invest more CPU resources when more metrics have to be processed. The memory consumption of the Node Exporter remains practically constant throughout the experiment, which shows that, similarly to the first experiment, the number of monitored metrics has no effect on the memory consumption. The consumed network bandwidth increases linearly but has a sharp drop at 1268 metrics. The right side of Figure 7 shows that at this point the system has reached its network limit regarding the number of monitored metrics (x-axis) that are transmitted to the server. Consequently, the failure rate (y-axis) jumps to almost 100 percent, which in turn causes the Node Exporter to not send any monitoring data in most of its HTTP responses. The scrape duration (y-axis) on the left side of Figure 7 shows an expected linear trend, which increases with the number of monitored metrics.

In summary, the experiments in this section have revealed two things. First, Prometheus is unable to handle high-demanding monitoring scenarios on low-end IoT devices without using an unfeasible share of CPU resources. Second, the tested monitoring solution with Prometheus was error-prone. In a typical stream processing scenario with edge devices, a monitoring framework needs to be able to handle overloaded nodes. However, under these circumstances, we either received no metrics at all or with a significantly increased delay. This shows that Prometheus cannot be efficiently used for monitoring IoT topologies. We expect other cloud-based monitoring solutions to exhibit similar behavior, since their architecture and functionality are similar to Prometheus as we described in Section 2.

4 The Road Ahead: Monitoring of Data Management Systems for the IoT

The approaches presented in Section 2 can be seamlessly implemented and integrated in cloud-based SPEs running on JVMs. There exist 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 [32]. Additionally, topologies comprising the cloud, fog, and edge are more complex than the purely cloud-based master/worker architectures. 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.

More concretely, the following properties distinguish IoT topologies from cloud-based ones:

1. **Resource constrained environment**: IoT devices typically have limited bandwidth and resources.
2. **Massive number of nodes**: IoT topologies scale up to many millions of nodes whereas cloud topologies have hundreds to thousands of nodes.
3. **Dynamic topology**: IoT topologies might be exposed to frequent changes, which occur due to the large amount of volatile and moving devices.
4. **Complex networks**: Devices in IoT topologies are often located in geo-distributed networks that encompass different access and security standards.
5. **Diversity**: IoT topologies usually comprise many devices with varying hardware and operating systems.
6. **Non-JVM-based**: IoT data management systems might not support JVMs.

Based on the above-mentioned properties and the functional requirements for general monitoring systems for the IoT proposed in [25], we draw 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 an IoT data management system needs to satisfy to enable monitoring in IoT topologies: 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 the complex networks property, and the last category addresses the properties diversity and non-JVM-based. In the remainder of this section, we describe the functional requirements for each of the identified categories.

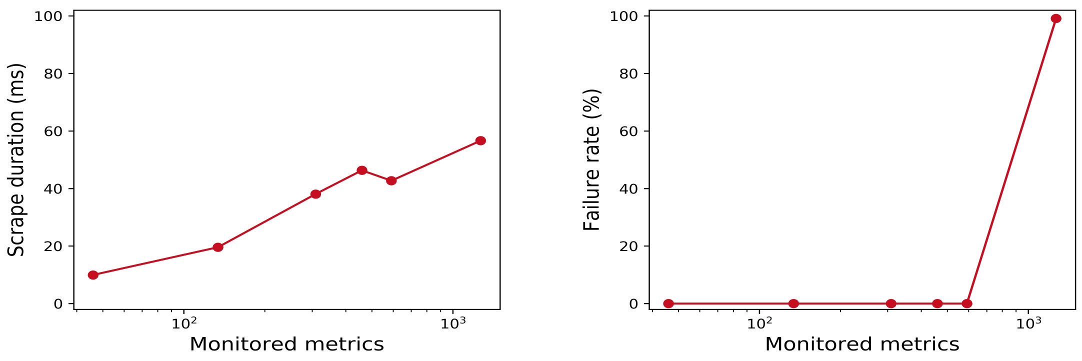


Figure 7: Scrape duration and failure rate of scrape jobs over different numbers of monitored metrics (sampling period = 100 ms)

4.1 Performance Optimization and Scalability

Data management systems for the IoT intend to distribute computation to millions of nodes and run thousands of queries in parallel [32]. In such a setting, the system requires monitoring data of many nodes to enable, for example, efficient optimization strategies and operator placement. The first category provides therefore a set of functional requirements that aim to reduce the amount of transmitted metrics and to optimize the CPU utilization of monitoring, thereby enabling efficient management of monitoring data.

Filtering of measured metrics: Metrics like CPU or memory often change significantly after the occurrence of certain events, e.g., a new operator placement. To avoid the transmission of redundant or insignificant changes, the monitored nodes have to provide threshold-based filtering, thereby reducing the communication overhead for monitoring data transmission and storage.

Adaptive sampling periods: Measurement intervals of metrics for different applications should be determined according to application-specific trade-offs. Small periods negatively affect the performance of the system, while large sampling periods diminish the accuracy of monitoring information. Therefore, custom time periods ranging anywhere from a few seconds to minutes, hours, or days need to be supported to monitor IoT data management systems.

Adaptive aggregation of monitoring events: IoT topologies consist of millions of nodes. Thus, metrics from a monitored node often have to pass multiple hops until they reach the final destination of the monitoring master. Aggregations at different hierarchical levels (e.g., locally or at intermediate levels) are required to avoid overhead and enable summarized views on subsets of the topology. Many existing monitoring systems already support metrics aggregation [18, 19, 27]. These systems require, however, to manually specify the aggregation points in the topology. IoT topologies change frequently and consist of such a scale that the aggregation points can not be maintained manually. Consequently, the monitoring system needs to determine automatically how and where to aggregate the metrics.

Adaptive CPU utilization control: Data streams change over time and thus the monitoring solution needs to be adapted accordingly, without affecting the performance of other components [8]. For example, in cases where the CPU utilization on a device is increased due to the deployment of new operators, the monitoring system needs to be able to reduce the overhead of gathering and processing performance metrics by appropriately dividing the processing resources between the components.

Learning from the past: A monitoring system for the IoT has to be adaptive as discussed in the previous paragraphs. Adaption strategies can be enabled by using historical data, e.g., inference techniques like machine learning can be utilized on historical data to avoid measuring and transmitting metrics. Consequently, a monitoring system for the IoT should be able to archive and retrieve monitoring data in a long-term manner to enable smart adaption strategies.

4.2 Handling Uncertainties

The edge of the network is a highly dynamic environment where the availability and location of devices change frequently. For time-critical edge computing applications, it is thus important that the monitoring solution can detect and collect information about the rapidly changing environment. To that end, the second category consists of functional requirements to handle highly dynamic topologies.

Discovery of changes: Changes regarding availability or geographical position are required for informing components to induce respective up- or down-scaling actions, e.g., new optimization actions regarding operator placement. Techniques of auto discovery and custom alerting are possible solutions for satisfying this requirement [28].

Adaptive data traffic control: Monitoring the network quality of connections in the edge can enable the beneficial adaption of the network communication. For instance, in situations where the network connections do not support high bandwidth, the amount of transmitted metrics needs to be reduced via decreasing the sampling rate or changing the network communication protocols.

Handling of failures: In a dynamic IoT topology, there are various kind of failures that can negatively affect the data management system. For example, 1) network partitions can prohibit the system from collecting accurate metrics, 2) back pressure can reduce the throughput and limit the performance of the system, or 3) bugs in the system might lead to unexpected behavior that can terminate running processing tasks. Therefore, the monitoring component needs to handle outages, deal with missing metrics, and track why and how problems occur in the system.

4.3 Permission and Access Control

Data management systems for the IoT are designed to run thousands of queries by a multitude of users in parallel. Users are located in different areas which differ in network properties, security settings and permission rights. Consequently, the third category describes the functional requirements to enable monitoring in a multi-tenant environment with different security standards.

Multi-tenant monitoring environment: The monitoring system needs to have the ability of defining multiple roles and views for various types of users with different permissions to access monitoring data. Different tenants should be able to measure parameters and gain access only to the information that pertains to them.

Network accessibility: Specific types of traffic such as Internet Control Message Protocol (ICMP) or Simple Network Management Protocol (SNMP) packets are filtered in private administrative domains due to firewalls or network rules because of security concerns. In such cases, the monitoring solution should automatically change its mode of operation via different communication protocols or alternative methods to collect metrics. For example, the location of a device can be tracked either by querying its geographical coordinates or via its IP address.

4.4 Handling Heterogeneity

An IoT data management system that supports real-time processing in a unified fog-cloud environment needs to operate independently from underlying cloud infrastructure providers, operating systems, and hardware. In the last category, we list the requirements to enable monitoring on different cloud providers and heterogeneous devices.

Extensibility: IoT data management systems need to support user-defined metrics. Therefore, it is necessary to support the customizability of monitoring solutions such as metric extension (i.e., incorporate and start measuring any new particular metric), which allows covering conditions particular to a specific component.

Support of different operating systems and devices: Monitoring solutions operating in the edge should cover all kinds of hardware virtualization and operating systems. For example, one way to collect performance metrics in a Linux environment is via cgroups, while in Windows with tools like the Windows Management Instrumentation (WMI) or the Windows assessment tool. Every operating system and device provides different ways to gather monitoring data, and a monitoring system for the IoT should be compatible with a large variety of devices and operating systems.

Standardized communication: A monitoring system exposes metrics about monitored entities. Therefore, it should use standardized communication protocols to enable efficient communication and compatibility between different endpoints. A widespread solution to expose monitoring metrics is, for example, via a RESTful API. For the IoT, there also exist low-overhead standardized protocols, like the Constrained Application Protocol (COAP) or IPv6 over Low power Wireless Personal Area Network (6LoWPAN).

In summary, IoT topologies consist of heterogeneous devices that form large and complex networks and undergo frequent changes. To address these properties, we identified four categories of functional requirements that are necessary to enable monitoring for a data management system for the IoT. These categories address the performance limitations and scale of the IoT, the inherent uncertainties, the multi-tenancy of the infrastructure, and the heterogeneity of the environment.

5 Conclusion

This paper explores existing monitoring solutions with respect to their applicability in a stream processing setting for IoT environments that contain millions of low-end devices. We described the architecture of SPE-external monitoring frameworks and provided an overview of SPE-internal monitoring components regarding three aspects: metric gathering, metric management, and integration with third-party systems. We identified that even though current solutions provide sophisticated methods to make monitoring easy-to-use and adaptive, they are tailored for cloud infrastructures and are not sufficiently efficient for IoT topologies with low-end devices.

We tested the monitoring system Prometheus and identified bottlenecks and inefficiencies in an IoT cluster with limited hardware resources. Our results indicate that Prometheus cannot handle high-demanding monitoring scenarios with fast sampling rates on edge computing devices and cannot support a high number of monitored metrics on those devices. In particular, with a decreasing sampling period and an increasing number of monitored metrics, the CPU usage quickly reached a level that is not acceptable for an IoT monitoring system.

We conclude that novel monitoring solutions are required to support the distribution, heterogeneity, volatility, and complexity of data stream processing in large-scale IoT environments. To that end, we identified four categories of functional requirements that need to be satisfied to enable monitoring of stream processing engines beyond the cloud.

Acknowledgements

We thank Lennart Behme for his assistance with the experiments. This work was supported by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and 01IS18037A).

References

[1] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl *et al.*, "The stratosphere platform for big data analytics," *The VLDB Journal*, vol. 23, no. 6, pp. 939–964, 2014.

[2] Alibaba Cloud, "Metrics principles and practices in flink," last accessed March 23, 2021, from https://www.alibabacloud.com/blog/metrics-principles-and-practices-flink-advanced-tutorials_596634/.

[3] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: Operator scheduling for memory minimization in data stream systems," in *ACM SIGMOD*, 2003, pp. 253–264.

[4] B. Babcock, M. Datar, and R. Motwani, "Load shedding for aggregation queries over data streams," in *IEEE ICDE*, 2004, pp. 350–361.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[6] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli, "Optimal operator placement for distributed stream processing applications," in *ACM DEBS*, 2016, pp. 69–80.

[7] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, "Operator scheduling in a data stream manager," in *VLDB*, 2003, pp. 838–849.

[8] A. Chaudhary, S. Zeuch, and V. Markl, "Governor: Operator placement for a unified fog-cloud environment," in *EDBT*, 2020.

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

[10] Dropwizard Team, "Dropwizard 2.1.0," last accessed April 1, 2021, from https://github.com/dropwizard/metrics/.

[11] Elastic, "Elastic ecosystem," last accessed June 14, 2021, from https://www.elastic.co.

[12] D. Giouroukis, A. Dadiani, J. Traub, S. Zeuch, and V. Markl, "A survey of adaptive sampling and filtering algorithms for the internet of things," in *ACM DEBS*, 2020, pp. 27–38.

[13] Grafana Labs, "Grafana 6.5.2," last accessed May 6, 2021, from https://grafana.com/.

[14] P. M. Grulich, B. Sebastian, S. Zeuch, J. Traub, J. v. Bleichert, Z. Chen, T. Rabl, and V. Markl, "Grizzly: Efficient stream processing through adaptive query compilation," in *ACM SIGMOD*, 2020, pp. 2487–2503.

[15] InfluxDB, "Influxdb 2.0," last accessed April 17, 2021, from https://docs.influxdata.com/influxdb/v2.0/.

[16] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *NetDB*, vol. 11, 2011, pp. 1–7.

[17] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *ACM SIGMOD*, 2015, p. 239–250.

[18] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817 – 840, 2004.

[19] Nagios Enterprises, "Nagios xi 5.6.10," last accessed May 6, 2021, from https://www.nagios.com.

[20] E. B. Ouro Paz, E. Tzirita Zacharatou, and V. Markl, "Towards Resilient Data Management for the Internet of Moving Things," in *Datenbanksysteme für Business, Technologie und Web (BTW)*, ser. LNI, vol. P-311, 2021, pp. 279–301.

[21] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *IEEE ICDE*, 2006.

[22] Prometheus Authors, “Prometheus 2.15.9,” last accessed May 6, 2021, from https://prometheus.io/.

[23] S. Rizou, F. Dürr, and K. Rothermel, “Solving the multi-operator placement problem in large-scale operator networks,” in *ICCCN*, 2010, pp. 1–6.

[24] S. G. H. Soumyalatha, “Study of IoT: understanding IoT architecture, applications, issues and challenges,” in *ICICN*, 2016.

[25] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, “Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review,” *Journal of Systems and Software*, vol. 136, pp. 19 – 38, 2018.

[26] The Netty project, “Netty 4.1,” last accessed April 29, 2021, from https://netty.io/.

[27] D. Trihinas, G. Pallis, and M. D. Dikaiakos, “Jcatascopia: Monitoring elastically adaptive applications in the cloud,” in *IEEE/ACM CCGrid*, vol. 14, 2014, pp. 226–235.

[28] C. N. Ververidis and G. C. Polyzos, “Service discovery for mobile ad hoc networks: a survey of issues and techniques,” *IEEE Communications Surveys Tutorials*, vol. 10, no. 3, pp. 30–45, 2008.

[29] J. Xu, Z. Chen, J. Tang, and S. Su, “T-storm: Traffic-aware online scheduling in storm,” in *IEEE ICDCS*, 2014, pp. 535–544.

[30] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, and et al., “Apache spark: A unified engine for big data processing,” *Commun. ACM*, vol. 59, no. 11, p. 56–65, Oct. 2016.

[31] S. Zeuch, A. Chaudhary, B. D. Monte, H. Gavriilidis, D. Giouroukis, P. M. Grulich, S. Bress, J. Traub, and V. Markl, “The nebulastream platform: Data and application management for the internet of things,” *CIDR*, 2020.

[32] S. Zeuch, B. Del Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, “Analyzing efficient stream processing on modern hardware,” in *PVLDB*, 2019.

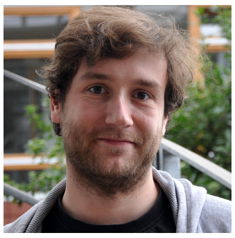
[33] S. Zeuch, E. Tzirita Zacharatou, S. Zhang, X. Chatziliadis, A. Chaudhary, B. Del Monte, D. Giouroukis, P. M. Grulich, A. Ziehn, and V. Mark, “Nebulastream: Complex analytics beyond the cloud,” *The International Workshop on Very Large Internet of Things (VLIoT)*, 2020.

[34] J. Zhou, P.-A. Larson, and R. Chaiken, “Incorporating partitioning and parallel plans into the scope optimizer,” in *IEEE ICDE*, 2010, pp. 1060–1071.

AUTHOR BIOGRAPHIES

Xenofon Chatziliadis is a Ph.D. candidate at the DIMA Group at TU Berlin. His research interests include distributed systems, performance monitoring, and IoT environments. He received his M.Sc. in Computer Science at TU Berlin.

Eleni Tzirita Zacharatou is a Senior Researcher at the DIMA Group at TU Berlin. She conducts research on mobility-aware data processing and spatial big data analytics and is further interested in robust stream processing in IoT environments. Her research results have appeared in premier data management venues (like VLDB, CIDR, ICDE), and her work has received the 2018 ACM SIGMOD best demonstration award. Eleni holds a Ph.D. in Computer Science from the École Polytechnique Fédérale de Lausanne (EPFL) and a Diploma - M.Eng. degree in Electrical and Computer Engineering from the National Technical University of Athens (NTUA).



Steffen Zeuch is a Senior Researcher at the DIMA Group (TU Berlin) and IAM Group (DFKI). His research interests are modern hardware, and the IoT. He published research papers on query optimization and execution as well as on novel system architectures. He did his Ph.D. in Computer Science at Humboldt University Berlin.

Volker Markl is a Full Professor and Chair of the Database Systems and Information Management (DIMA) Group at TU Berlin, Chief Scientist and Head of the Intelligent Analytics for Massive Data Research in DFKI, and Director of the Berlin Institute for the Foundations of Learning and Data (BIFOLD). He has published numerous research papers on indexing, query optimization, lightweight information integration, and scalable data processing. He is a Fellow of the ACM as of 2021.