The NebulaStream Platform: Data and Application Management for the Internet of Things

Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Bress, Jonas Traub, Volker Markl

The NebulaStream Platform: Data and Application Management for the Internet of Things

Steffen Zeuch[1,2] Ankit Chaudhary[1] Bonaventura Del Monte[1,2] Haralampos Gavriilidis[1]
Dimitrios Giouroukis[1] Philipp M. Grulich[1] Sebastian Breß[1] Jonas Traub[1,2] Volker Markl[1,2]

[1]Technische Universität Berlin [2]DFKI GmbH

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 highlight these major challenges and outline how existing systems handle them. To address these challenges, we introduce the NebulaStream platform, a general purpose, end-to-end data management system for the IoT. NebulaStream addresses the heterogeneity and distribution of compute and data, supports diverse data and programming models going beyond relational algebra, deals with potentially unreliable communication, and enables constant evolution under continuous operation. In our evaluation, we demonstrate the effectiveness of our approach by providing early results on partial aspects.

1. INTRODUCTION

Over the last decade, the amount of produced data has reached unseen magnitudes. Recently, the International Data Corporation [43] estimated that by 2025 the global amount of data will reach 175ZB and that 30% of these data will be gathered in real-time. In particular, the number of IoT devices is expected to grow to as many as 20 billion connected devices by 2025 [21]. At the same time, devices such as embedded computers or mobile phones continuously increase their processing capabilities. This trend enables the exploitation of their computing and communication capabilities, as they become objects of common use. As a result, the IoT is one of the fastest emerging trends in the area of information and communication technology [34].

The explosion in the number of connected devices triggers the emergence of novel data-driven applications. These applications require low-latency, location awareness, widespread geographical distribution, and real-time data processing on potentially millions of distributed data sources. To enable these applications, a data management system needs to leverage the capabilities of IoT devices.

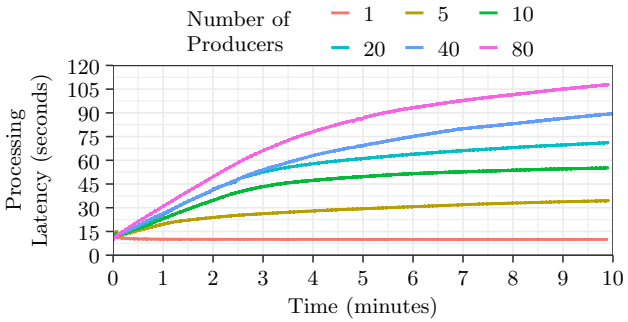


Figure 1: IoT application using a cloud-centric SPE.

However, today's data management systems are not yet ready for these applications as they embrace either the cloud or the fog computing paradigm. Systems based on the cloud paradigm, e.g., Flink [2], Spark [55], and Kafka Streams [45], do not exploit the full capabilities of IoT devices. To implement IoT applications, these systems require the collection of sensor data centrally in a data center prior to applying processing. This centralized processing paradigm presents a bottleneck for upcoming IoT applications, which need to process data from millions of distributed sensors.

In Figure 1, we showcase the impact of this bottleneck by executing an IoT application scenario using a cloud-based approach and reporting the average processing latency. To this end, we scale the number of IoT data producers from 1 to 80. Each producer generates data at a constant speed of 50K record/sec. Producers send their data over a gateway to an Kafka cluster with five nodes. Inside the same cloud environment, we setup an Flink cluster with eight nodes (cloud nodes are connected through a 1 Gbit Ethernet connection). Our Flink job reads data from Kafka and executes a tumbling windowed aggregation of 10 seconds to count distinct events. We let the experiment run for 10 minutes and measure the end-to-end processing latency following the methodology introduced by Karimov et al. [24]. Our experiment shows that latency increases as we increase the number of producers. Our cloud-based IoT application scenario can sustain up to 20 producers with constant latency. Beyond this point, our application saturates and latency increases gradually. This effect intensifies for more IoT producers and results in a continuously increasing backlog within Kafka. Overall, our experiment shows that a centralized cloud approach does not scale for IoT applications and thus future IoT applications require a new system.

10th Biennial Conference on Innovative Data Systems Research (CIDR '20) January 12-15, 2020 , Amsterdam, Netherlands.

In contrast, systems based on the fog computing paradigm, e.g., Frontier [37] and CSA [48], exploit the processing capabilities of edge devices, i.e., devices that are physically closer to the data sources. These devices apply data reduction techniques, e.g., pre-selection or pre-aggregation, to reduce data volume as early as possible in the processing pipeline, i.e., close to the sensor. However, fog computing systems only scale within the fog and do not exploit the virtually unlimited resources of modern cloud infrastructures (e.g., Amazon Web Services or Microsoft Azure).

Data management systems for wireless sensor networks (WSNs), e.g., TinyDB [30], exploit small battery-powered sensors to create a network of nodes to capture physical phenomena, such as earthquakes or volcanic eruptions. These systems apply acquisitional query processing techniques to optimize the execution for battery lifetimes and deploy a small set of specialized queries to capture the physical phenomena. However, WSN systems only scale within the sensor networks and do not exploit the resources of the attached cloud and fog environments. In particular, they do not consider offloading computation to external nodes and do not provide general-purpose query execution capabilities.

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 such as Flink or Spark. To enable future IoT applications, 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 individual advantages and enable cross-paradigm optimizations (e.g., fusing, splitting, or operator reordering). From a system point of view, this unified environment imposes three unique characteristics that are not supported by state-of-the-art data management systems.

Heterogeneity: A unified environment consists of a highly heterogeneous hardware landscape. The processing nodes range from low-end battery-powered sensors (e.g., Mica Motes) over system-on-a-chip devices (e.g., Raspberry PIs) to high-end rack-scale servers. In particular, cloud infrastructures consist of homogeneous node setups, whereas the fog contains heterogeneous, low-end computing devices. Furthermore, WSNs consist of highly specialized battery-powered sensors. To exploit the individual capacities of each node, an IoT data management system has to take their individual capabilities into account, especially their resource restrictions. However, current data management systems abstract from the underlying hardware with virtual machines and managed runtimes. These abstractions hinder the exploitation of specialized instructions and processing units and prevent important optimizations.

Unreliability: A unified environment has to handle different levels of runtime dynamics. The fog introduces a highly dynamic runtime environment with unreliable nodes that might change their geo-spatial position, i.e., resulting in many transient errors or changes in latency/throughput. WSNs exacerbate this highly dynamic runtime even further by turning-off sensors temporally to save energy and allowing reads only following a dedicated read schedule. In contrast, a cloud infrastructure is a relatively stable environment where node failures are rare. However, current approaches for load balancing, fault-tolerance, and correctness only concentrate on one particular environment. Thus, these approaches miss out important cross-paradigm optimization potential.

Elasticity: In a unified environment, data move from the sensors via intermediate nodes to the cloud, and finally to the consumer, e.g., a user device or another system. The fog topology is commonly built as a tree-like network topology [10, 19] with several dataflow paths. Data processing in the fog topology has to be network-aware because only nodes on the path from the sensors to the cloud can participate. Furthermore, in a WSN, all sensors send their data to the next sensor in range until all data end up at the root of the network. In contrast, in the cloud, every node has access to all data, e.g., via a distributed file system, e.g., HDFS. However, current approaches allow optimizations, scaling, and load balancing only within nodes of the same environment and thus miss out important cross-paradigm optimization potential.

Overall, a unified environment introduces a previously unprecedented, unique combination of characteristics, i.e., 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 over millions of sensors.

In this paper, we propose *NebulaStream* (NES), a novel data processing platform that addresses the above-mentioned heterogeneity, unreliability, and scalability challenges and enables effective and efficient data management for the IoT. In particular, NES copes with these unique characteristics as follows. First, NES copes with heterogeneity by maximizing *sharing of results* and *efficiency of computing* to significantly reduce the amount of data transferred and to exploit hardware capabilities efficiently. Second, NES addresses unreliability by applying *dynamic decisions* and *incremental optimizations* during runtime to be as flexible as possible. Third, NES enables elasticity by designing each node to react *autonomously* to a wide range of situations during runtime. With NES, we enable future IoT applications by unifying sensors, fog, and cloud in one general-purpose, end-to-end data management platform. Our early experiments show that NES reduces the amount of data and sensor reads up to 90%, increases node throughput and decreases energy consumption on low-end devices by up to two orders of magnitude, and processes queries with low latency even in the presence of many node failures.

The remainder of the paper is structured as follows. We show a typical IoT application scenario in Section 2. In Section 3, we describe the NebulaStream platform, discuss its design principles, and provide initial performance results. Finally, we survey related work in Section 4 and conclude in Section 5.

2. IoT APPLICATION SCENARIO

In Figure 2, we present an integrated public transport system of Berlin as a representative IoT application scenario. The components in this scenario are either stationary or mobile. Vehicles (red and yellow boxes), i.e., taxis, buses, subways, and trains move around the city and carry a set of sensors and a simple processing unit. Each unit collects vehicle data (e.g., routing, maintenance information, and occupancy/usage) as well as data from the environment (e.g., traffic, road conditions, and weather). The base stations, processing nodes, and dispatch station are stationary components. Base stations (green triangles) are distributed across the city and consist of antennas, network routers, and compute and storage capacity. Processing nodes (green circles) are dis-

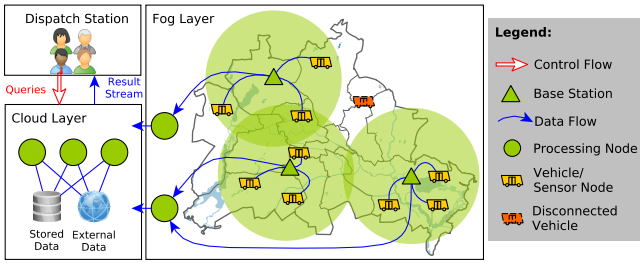


Figure 2: IoT application scenario.

tributed within the city to gather data from several base stations and apply more complex processing. The centralized dispatch station represents the endpoint for all data and merges data from the fog and the cloud with stored and external data. Users manage public transport through the dispatch station. This IoT scenario requires a massively distributed system with continuous data producers as well as transient and permanent, distributed compute and storage capabilities.

The environment in this scenario differs fundamentally from current cloud-based data processing architectures. In particular, vehicles move within the city and interact with multiple antennas, which transmit data to base stations. Due to the dynamic nature, vehicles may encounter temporary connection losses or outages (red vehicle), e.g., when they are outside of transmission ranges. Furthermore, all vehicles move at different speeds, on different roads/tracks, and are potentially equipped with different hardware. User queries addressing only a subset of the vehicles do not require collecting all sensor data from all vehicles at every transmission interval. This represents a major characteristic that is crucial for enabling large-scale IoT applications. As a result, a fog requires continuous adaptation to a dynamic environment with respect to faults and changes in the availability, amount, type, capacity, and location of data and compute nodes. Furthermore, on the sensor level, a system has to continuously adapt the sensor reads depending on a dynamic query workload.

Despite the distributed nature, it must be possible to manage the system through a centralized, global view and execute continuous as well as ad-hoc data analytics. This includes the entire data analysis pipeline, from information extraction to integration and model building using machine learning, signal processing, and other advanced analytics.

From a user perspective, this system may assist the public transport dispatcher to schedule new vehicles or reroute vehicles in case of outages or increased passenger demand. This results in a feedback loop that may change the physical fog architecture. Furthermore, this architecture allows for enriching real-time data with external sources, e.g., air pollution measurements, event calendars, area crowdedness, or knowledge bases. The characteristics of this application are representative for many IoT scenarios including Industry 4.0, smart homes, smart grids, smart cities, or participatory sensing applications.

3. NEBULASTREAM PLATFORM

In this section, we present the NebulaStream (NES) platform. First, we describe the common topology of IoT application scenarios and highlight its novelty (Section 3.1). After that, we identify key design principles for an IoT data management system (Section 3.2) and later describe how

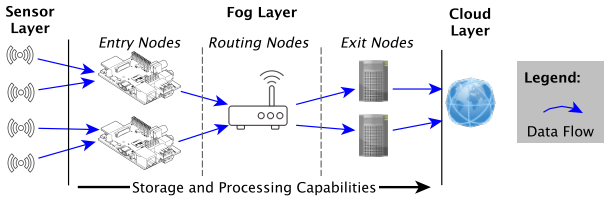


Figure 3: Multi-layer NES Topology.

NES implements them (Section 3.3). Finally, we discuss challenges for an IoT data management system and how NES addresses them (Section 3.4).

3.1 NES Topology

In Figure 3, we present a multi-layer NES Topology that is common in today's IoT infrastructures [10]. This figure presents the dataflow from the sensors to the cloud. The basic assumptions in this topology are three-fold. First, all data might reach the *Cloud Layer*. Second, devices on the path from the sensors to the cloud are able to apply processing. Third, the Cloud Layer is able to apply remaining processing, i.e., representing a fall-back mechanism. In contrast, all other nodes can only access data if they are routed through them and their storage and processing capabilities determine the operations they can apply.

The data are routed among the three layers as follows. On the *Sensor Layer*, millions of sensors produce data without processing them. However, NES is able to schedule the sensor reads depending on the query, e.g., increasing read frequency or omitting reads. Sensors provide two data access patterns: pull-based and push-based. Each sensor is connected to at least one low-end node in the *Fog Layer*, which is responsible for this sensor (so-called *Entry Node*). In the *Fog Layer*, NES processes data as they flow from Entry Nodes to Exit Nodes. During processing, nodes may change their geo-spatial position. The data transfer is orchestrated by *Routing Nodes*, such as routers or switches. The data processing capabilities on Routing Nodes are restricted and the provided functionality is highly vendor-dependent [4, 29]. In general, the storage and processing capabilities of nodes increase significantly in the NES Topology with each hop towards the Cloud Layer. After leaving the Fog Layer through an *Exit Node*, data enter the Cloud Layer. The Cloud Layer provides virtually unlimited scaling of compute and storage. In IoT application scenarios, this layer will perform the remaining computation and output the data to the user. An alternative approach to this centralized design would allow each node in the fog to function as a potential sink. Thus, users would submit their queries directly through their device and each device would represent an exit node in the topology. In this decentralized design, each device will be responsible for answering the submitted user query. This design naturally supports geo-spatial query processing as most users are potentially only interested in data produced nearby. Exploring the design space of a centralized vs. a decentralized design is one major future challenge.

The NES Topology introduced in Figure 3 represents a fundamentally new and unique set of characteristics and requirements compared to common cloud infrastructures. First, query processing and operator placement have to be network-aware. The main query optimization goal is to find an efficient route through the Fog Layer that reduces data

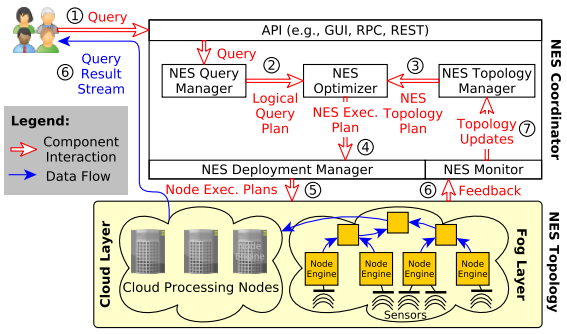


Figure 4: NES architecture overview.

volumes as early as possible without violating any Service-Level-Agreement (SLA) but fulfilling Quality of Service (QoS) constraints. Second, the NES Topology is highly heterogeneous and many nodes have only limited processing capabilities. In particular, nodes in the lower parts of the Fog Layer are restricted in storage and processing capabilities. Furthermore, processing has to trade-off between energy consumption and performance. Third, the Fog Layer is highly unreliable compared to the homogeneous and relatively stable Cloud Layer. To support mobility and related aspects, the system has to take the characteristics of each individual environment into account. Fourth, the volume and velocity of sensor data represent an external factor. As a result, the entire system has to evolve around sensor data that is injected by the outside world. With NES, we build a platform that creates a federation of sensors, fog, and cloud, which enables big data acquisition and analysis.

3.2 NES Design Principles

NES is a platform for future IoT applications that copes with the unique set of characteristics of a unified environment. For individual layers, different approaches were proposed over the last decades. However, combining all of them into a single system is the major challenge that we address with NES. To handle millions of sensors and thousands of queries, we base the system design of NES on the following design principles:

1. **Dynamic Decisions:** NebulaStream never expects a static behavior or conditions in any component.
2. **Autonomous Processing:** NebulaStream equips compute nodes with all logic necessary to act as autonomously as possible.
3. **Incremental Optimizations:** NebulaStream optimizes a network of active queries in incremental steps rather than traditional query optimization or batched changes.
4. **Maximize Sharing:** NebulaStream shares data and processing wherever possible, i.e., on windows (stream slicing), among queries (multi-query optimization), on sensor data (acquisitional query processing), and on operator level (code optimization).
5. **Maximize Efficiency:** NebulaStream applies hardware-tailored code generation to exploit the underlying hardware efficiently.
6. **SLA Centric Processing:** NebulaStream's primary goal is to match user-provided SLAs and QoS constraints with available resources.
7. **Ease of Use:** NebulaStream enables users to choose their preferred programming environments and models, without worrying about system-internals and performance implications.

3.3 NES Architecture

In Figure 4, we present the architecture of NebulaStream. In general, we design NES with a centralized deployment process and a decentralized run-time re-optimization. In particular, we envision a *logically* centralized deployment process in which one central instance has control over the deployment. However, this logically centralized instance can be distributed among multiple region coordinators to form a hierarchy of coordinators. In the future, we envision moving towards a decentralized deployment process that enables every device to timely submit queries and receive results. In the current design, users interact with NES through one of the provided APIs to send queries to the *NES Coordinator* ①. Our current APIs allow specifying dataflow programs, similarly to the APIs of streaming systems like Flink, Spark, and Storm. The NES Coordinator consists of several components that orchestrate query processing. The *NES Query Manager* is responsible for creating logical query plans from user requests ②. Additionally, this component maintains *logical streams* that represent logical views over sensors, e.g., a logical stream *cars* could combine sensor inputs from multiple cars into one consistent stream. The *NES Topology Manager* orchestrates the NES Topology, which consists of workers and sensors. During startup, each device registers itself and provides information, such as resource capabilities and network topology information. However, to reduce the complexity of optimization decisions, NES follows the idea of introducing *zones* that aggregate a sub-tree or geo-spatial region of the topology into one node. Thus, the optimizer treats a zone as one node which transparently abstracts from the dynamic behavior inside the zone. As a result, a topology may consist of a hierarchy of zones, which simplifies the global optimization process. The efficient assembly of zones is one future research challenge for NES.

The *NES Optimizer* provides the assignment of a logical query plan (created by the NES Query Manager) to the current NES Topology plan ③ (maintained by the NES Topology Manager). This assignment defines the *NES Execution Plan (NES-EP)*. The assignment process introduces a large optimization search space, e.g., operators can be assigned top-down, bottom-up, or by other assignment strategies. The *NES Deployment Manager* takes the NES-EP ④, disassembles it into Node Execution Plans (Node-EPs), deploys them to the nodes in the NES Topology, i.e., into either the Fog or the Cloud Layer, and sets up the sensors ⑤. This deployment is performed incrementally and requires rerouting data on different dataflow paths. Note that this deployment process has to handle a gap between optimization and deployment time. Thus, optimization is based on a snapshot of the topology, while deployment has to take the current topology into account. Therefore, the deployment process in this highly dynamic execution environment introduces many interesting research challenges, such as the partial deployment of plans and the partial re-optimization of sub-plans. The *NES Monitor* constantly collects feedback from the NES Topology ⑥ and maintains statistics and current resource utilization for the NES Topology Manager ⑦. To improve operator placement, the NES Optimizer requests these statistics and current resource utilization from the *NES Monitor* ⑦. However, maintaining a centralized, coherent view over a large and highly dynamic topology is a major research challenge. First, the NES Optimizer has to be aware that the topology data is potentially

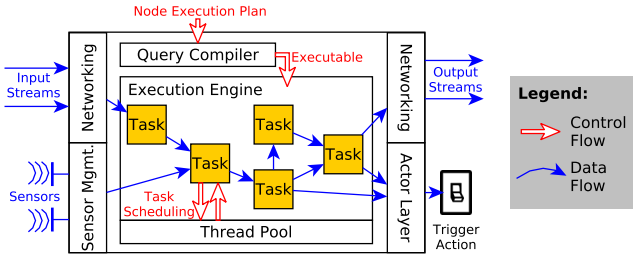


Figure 5: NES Node Engine.

out-dated and thus has to optimize accordingly, e.g., by providing a set of alternative plans. Second, the collection of monitoring data and the maintenance of statistics has to take the current system load into account and thus must be prioritized lower than data transfers to answer user queries. Third, we envision a decentralized run-time re-optimization process that is triggered by the nodes themselves. To this end, NES nodes first attempt to address a change locally, then communicate with their neighboring nodes, and finally requesting support from a central coordinator.

In Figure 5, we show the components of the node engine, which is deployed on all devices of the NES Topology. The *NES Node Engine* is responsible for communicating with the NES Coordinator, accepting Node-EPs and control messages, as well as setting up the input sources, output sinks, and other components. The incoming queries are Node-EPs, which contain a partial subtree of the overall NES-EP. The Node-EP is compiled by the local query compiler and later injected into the processing tasks. As input, the NES Node Engine receives data from the network, e.g., from another node, or directly from an attached sensor. As output, the NES Node Engine either sends data over the network or triggers an action on an attached device, e.g., controlling an actuator such as a light switch.

The *Execution Engine* orchestrates the processing inside each NES Node Engine. The central unit of work is one task that combines n input buffers, m output buffers, and the execution of the specified operators [57]. The processing in NES is *source-driven* and applies the following sequence of steps on each incoming buffer. First, the engine assembles the tasks by embedding the executable and allocates all required input, intermediate, and output buffers. After that, the engine enqueues the tasks in one of the processing queues. Finally, each thread in the *Thread Pool* dequeues one task, processes it, and either enqueues the result buffer into an output queue or triggers an action. This highly dynamic design enables high resource utilization but also introduces a dynamic execution order, which poses new challenges for the system design.

In addition to processing components, each NES Node Engine contains dedicated components for local and neighboring optimizations, windows, routing, sensors, state, and run-time re-optimization. As a result, we drastically reduce the complexity of the query compiler and increase maintainability and separation of concerns in NES. In particular, NES compiles only the *hot* code fragments and links other functionalities as pre-compiled components (following Neumann et al. [36]). Overall, it is a design decision in NES to equip the NES Node Engine with all necessary components to enable it to be as autonomous as possible. In particular, we assign all means to the node to enable it to make as many decisions as possible decentrally and independently. This

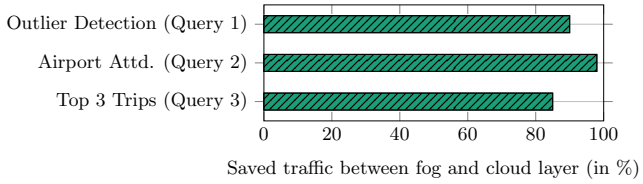


Figure 6: NES data reduction on the sensor level.

design follows the Borealis design [1] and tries to encounter transient changes locally and permanent changes globally. In NES, we envision a system design with autonomous nodes and a simple coordinator to mitigate potential bottlenecks in large scale environments.

3.4 NES Solutions for IoT Challenges

Based on the unique characteristics highlighted in Section 1 and IoT application scenarios presented in Section 2, we outline five main challenges for an IoT data management system. In the following, we discuss the challenges and propose our solutions.

3.4.1 *C1 - Heterogeneity, Distribution, and Volume of Data At-Rest and Data In-Motion*

NebulaStream's goal is to scale to thousands of queries and millions of sensors. In the IoT, data are generated by many distributed sources such as sensors or streams of other systems. A particular challenge originates from handling the sheer amount of diverse data sources, potentially up to the number of millions. These sources differ in their characteristics, ranging from millions of small sensor streams to a few large streams from sources such as click-streams or auctions. The accessibility of sources under security and privacy constraints, as well as efficient access paths, requires solutions completely different from what today's big data processing systems provide. For example, an IoT infrastructure enables new solutions for security and privacy as it allows local preprocessing of data next to the generation, e.g., inside a house or building. This enables a scenario where only authorized or anonymized data are sent to the central cloud. As a result, we can enable users to have full control of their own data. Overall, these characteristics imply research questions with respect to scalability, efficiency, integration, security, privacy, and interoperability.

To support this extreme diversity in NES, we follow the *Maximize Sharing* design principle (Section 3.2) and apply data sharing techniques on three different levels. First, on the query level, NES exploits data sharing among multiple streaming queries as proposed by Karimov et al. [25]. Second, on the operator level, NES slices data streams and exploits data sharing on stream aggregations as proposed by Traub et al. [50]. Third, on the sensor level, NES applies *Acquisitional Query Processing* (ACQP) [30] and *On-Demand Scheduling* of sensor reads and data transmissions [49]. These techniques limit data acquisition to data points which are required for answering user queries. By combining the introduced techniques in NES, we attempt to drastically reduce the amount of acquired, transferred, and processed data; thus, enabling IoT applications with thousands of queries over millions of sensors.

Figure 6 presents an initial experiment that demonstrates the potential savings of data reduction techniques in NES on the sensor level. We use the New York taxis data set [46], derive routes for each taxi trip, and replay the routes of all

taxis on Raspberry Pis, which represent sensor nodes located in taxis. As a baseline, we use a common IoT setup where sensor nodes stream current values to a central SPE in the cloud, without any knowledge about the executed queries. In contrast to this cloud-centric IoT setup, NES combines cloud and fog nodes as well as sensor nodes in taxis in one system to allow for holistic optimizations.

We show three example queries in an SQL-like notation. The queries include an outlier detection (Query 1), an airport attendance monitoring (Query 2), and a top three query for the longest ongoing trips (Query 3).

```
SELECT ts, medallion, trip_id, latitude,
    longitude, distance, passenger_count
FROM stream(taxis, 2000)
WHERE journey_flag=TRUE &&
    (latitude<40.249448 || latitude
    >41.381560 || longitude<-74.820611 ||
    longitude>-71.848319 || distance=0 ||
    passenger_count=0); -- NY area
```

Query 1: Journeys leaving the New York area and journeys without passengers. Checked every 2 seconds.

```
SELECT ts, sum(passenger_count)
FROM stream(taxis, 5000)
WHERE (40.536532<latitude AND latitude
    <40.745906)  && (-73.946390<longitude
    AND longitude<-73.609759) --airport
GROUP BY ts AHEADLIMIT 100 DELAYLIMIT 100;
```

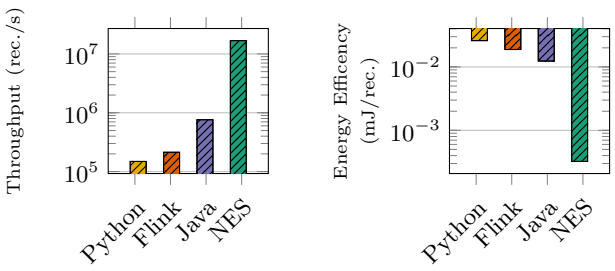
Query 2: Returning the number of passengers in the airport zone. Updated every 5 seconds.

```
SELECT ts, latitude, longitude,
    trip_distance
FROM stream(taxis, 1000)
WHERE journey_flag = TRUE
ORDER BY trip_distance DESC LIMIT 3;
```

Query 3: Returning the top three longest ongoing trips. Updated every second.

We modify the data acquisition process for all three queries such that only required data are sampled and transmitted. In particular, we can interleave data gathering operations (i.e., sensor reads) with data processing (e.g., filters) [30]. Theoretically, the system has to read all sensors specified in the *select* clause at the frequency specified in the *from* clause. However, the filter predicates in the *where* clause allow for preventing sensor reads and data transmissions for tuples that are filtered out. For instance, in Query 1 and Query 3, we first check the journey flag. If the value is `false`, we do not read any other sensor.

Another important optimization is to adjust sampling rates continuously and to prevent data transmissions based on the observed sensor values [49]. For example, in Query 1 and Query 2, we can use the current position of the taxi to calculate the earliest time when the taxi could leave New York or enter the airport area. Thus, we know upfront that no tuple will pass the filter for that time span and do not have to read or evaluate sensor values for that time. In addition, in Query 2, we specify a tolerance for sensor read times (*ahead* and *delay* limit), which saves data transmissions when multiple queries request values from the same sensor. We apply user-defined sampling functions to adjust sampling rates continuously, apply read time tolerances, and schedule sensor reads, respectively [49]. In Figure 6, we show that the saved traffic between the fog and the cloud layer is significant for all queries using these optimizations.



(a) Throughput. (b) Energy Efficiency.

Figure 7: YSB on RaspberryPi 3B+.

3.4.2 C2 - Heterogeneity, Distribution, and Volume of Compute

NebulaStream's goal is to exploit the hardware resources of millions of heterogeneous devices efficiently. A particular challenge originates from the potentially millions of compute devices that are found in a fog topology. These devices have a diverse set of capabilities, with respect to storage, processing, and interconnect. The devices range from small battery-powered sensors with no compute capabilities (beyond simple filtering) and an unreliable temporary connection to a large compute cluster with huge storage, infiniband interconnect, and thousands of compute cores. These characteristics imply challenges with respect to security, permission management, and efficient and effective resource utilization.

To support this heterogeneity in NES, we follow the *Maximize Efficiency* design principle (Section 3.2). In particular, we apply two techniques. First, we use query compilation, the leading paradigm for achieving high resource utilization in data-at-rest processing [36]. In NES, we transfer this approach to the special semantics of fog and stream processing. In particular, NES generates specialized code depending on the actual query, hardware, and data characteristics [56]. Second, NES distributes query optimization and code generation between the central coordinator and the local node engine. On the coordinator, NES performs global query optimizations (e.g., operator reorder) and splits the query into segments for individual devices. On the node engine, the query compiler produces hardware-tailored code to exploit the availability capabilities most efficiently.

Our experiment in Figure 7 evaluates the throughput and energy efficiency of the Yahoo Streaming Benchmark (YSB) on a RaspberryPi 3B+ using Python, Flink, a hand-optimized Java program, and NES, respectively. The YSB simulates a real-word stream processing task and consists of a filter and a windowed aggregation [12]. We implement the YSB with a one second tumbling window and 10000 campaigns based on the codebase provided by Gier et al. [16]. Our results show that hardware-tailored code generation is essential to efficiently utilize resources, especially for low-end devices. In Figure 7a, we present the maximal throughput of the four different YSB implementations. NES outperforms all other systems by at least 10x and is the only system that is able to reach a throughput of more than 10 million tuples per second. All other systems suffer from the high-overhead of the underlying managed runtime. This overhead is signif-

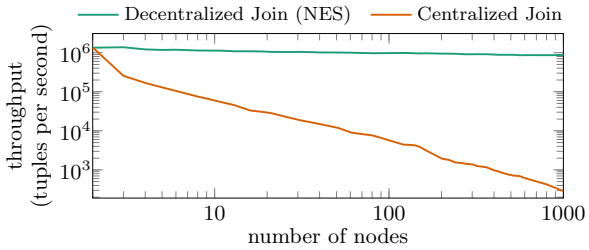


Figure 8: Gathering coherent snapshots from sensor nodes.

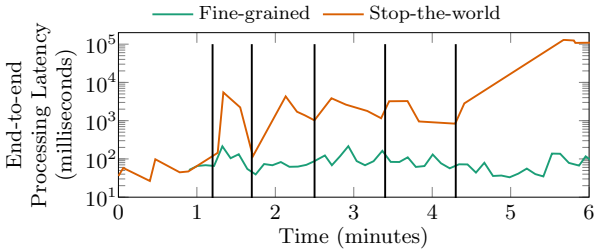


Figure 9: Evaluation of fault tolerance mechanisms.

icant on low-end devices like the RaspberryPi. Furthermore, through code generation, NES reduces the energy consumption per device and thus requires less energy to achieve the same performance. In Figure 7b, we evaluated the energy efficiency of the four different YSB implementations. To this end, we define energy efficiency as the required energy in milli joule per processed record. Our results show that NES requires around 0.0003 milli joule per tuple, which is an 80x improvement compared to the Python implementation. In the future, we will further investigate the trade-off between energy consumption and performance as one major research question for NES. Especially for battery-powered sensors, code generation enables a higher operation time and thus reduced maintenance and replacement costs.

As a second technique, we utilize in-network processing inside the Fog Layer to reduce the computation required at the Cloud Layer. In Figure 8, we present an example query that gathers values from up to 1000 nodes and joins them to coherent snapshots. A snapshot is coherent, if all sensor values contained in the snapshot have been read at the same time. In practice, snapshots are often incoherent, because the times of sensor reads are not perfectly aligned among all distributed nodes. In addition, clock deviations among sensor nodes lead to undetected incoherence, which potentially causes application failures such as false correlations.

We use the techniques which were introduced in the SENSE System [51] to ensure scalability and to mitigate incoherence. SENSE arranges sensor nodes in data gathering pipelines, which join tuples incrementally (decentralized join) and ensure coherence. In contrast to a centralized join, the Cloud Layer only joins the results of the pipelines instead of all individual sensor measures. This prevents a central bottleneck at the Cloud Layer and ensures high throughput when gathering values from a large number of sensors. As shown in Figure 8, a centralized join causes a drastic throughput decay when the number of nodes increases. In contrast, by utilizing the available computing resources on the path from the sensors to the Cloud Layer, NES achieves almost constant throughput and addresses coherence issues.

By applying hardware-tailored code generation and in-network processing, NES exploits the available compute resources most efficiently and allows for balancing computational demands and energy consumption.

3.4.3 C3 - Spontaneous, Potentially Unreliable Connectivity between Data and Compute

NebulaStream's goal is to detect and compensate potentially unreliable nodes in the Fog and Sensor Layer without impacting consistency and availability. A particular challenge originates from the need to manage data and compute together, as most applications will consist of ad-hoc or standing streaming queries. Furthermore, some compute units may be connected via Wifi, mobile, or satellite networks with intermittent connectivity and unreliable connections. In contrast to a homogeneous and relatively stable cloud environment, a heterogeneous and volatile fog environment has to handle frequent transient failures. Furthermore, WSNs are even more prone to transient failures due to their battery-powered low-end devices and vulnerable radio transmission.

Failures in the fog and in WSNs occur due to numerous reasons, most notably hardware errors, software errors, congestion that results in back-pressure (straggler nodes), inadequate resource allocation, and transient connection loss. Furthermore, devices continuously refresh their connections while moving and create ad-hoc connections that result in an unpredictable communication pattern [34]. This requires special solutions to deal with the intermittent availability of resources, both with respect to data and code management. The resulting challenges require changes in areas such as adaptivity, synchronization across devices, consistency, transaction management, recovery, and fault-tolerance.

Common cloud-centric SPEs handle node failures using a stop-the-world recovery protocol [11, 7]. When an error occurs, the system stops the entire processing and redeploys a new query plan. In contrast, NES adopts a fine-grained recovery protocol, i.e., NES restarts only the operator instances involved in a failure. To assess the performance of both protocols, we implement them in Flink and run the comparison on a simulated IoT environment. This environment comprises of 8 servers, which are equipped with Intel Xeon E5620 CPUs, 32 GB of RAM, and an 1 Gbits network. In Figure 9, we show the end-to-end processing latency of both protocols while randomly terminating compute nodes (indicated by the black vertical lines). As shown, the stop-the-world protocol cannot recover from high transient error rates as the latency constantly increases. In contrast, the fine-grained recovery protocol restarts failed operators without halting the entire query.

To achieve reliability in an unreliable environment, we apply the *Dynamic Decisions*, *Autonomous Processing* and *Incremental Optimizations* design principles in NES. Because a central component cannot keep up with the pace of failures in a dynamic environment, we apply a diverse set of techniques in NES. On each layer of the NES Topology, we apply different failure recovery approaches; thus, providing different guarantees. On the Sensor Layer, NES substitutes missing sensor values from broken sensors with nearby sensors, if applicable, or buffers the values during transient connection loss [30]. On the Fog Layer, NES extends the Frontier approach [37], which sends data through multiple network paths to achieve fault-tolerance. Furthermore, data are buffered by upstream operators and replayed in case of an error. On the Cloud Layer, NES extends existing fault-tolerance approaches, e.g., global checkpointing and message broker with fine-grained operator reconfiguration [13].

By extending and combining existing approaches on different levels of the NES Topology into a unified fault-tolerant solution, we attempt to handle spontaneous, potentially unreliable connectivity of IoT infrastructures.

3.4.4 C4 - Diversity in Programming and Management Environments

NebulaStream's goal is to support a diverse set of data processing workloads specified in different query languages and following different processing models (e.g., relational, linear, or graph algebra). A particular challenge originates from IoT applications that require a combination of different data-oriented programming paradigms. Possible workloads range over the entire data management pipeline, from information extraction over information integration to model building and inference. In particular, running AI/ML/Data Science algorithms in the fog enables direct feedback loops between the digital and the physical world. These workloads include potentially iterative algorithms mixing relational, linear, and graph algebra, and may run on top of continuous data streams or finite data sets. This diversity presents challenges with respect to 1) holistic, optimizable, intermediate representations, 2) efficient and scalable physical operators across all paradigms that can be mixed and matched, and 3) the combination of domain-specific and generic query languages that offers a sufficiently powerful yet optimizable interface to a data engineer. Furthermore, the programming and reasoning about sensors and actuators in such a distributed, diverse setting entails a huge challenge with respect to both, scalability and ease of use.

To support diverse workloads in NES and create a large community with diverse users from different fields, we envision an *easy-to-use* interface. In particular, we attempt to allow users to choose their preferred programming environments and models without the need to take system-internals and performance implications into account. To enable this diversity, we build on top of existing frameworks, such as Weld [40], Arc [27], Emma [3], and LARA [28] to represent diverse queries in a unified intermediate representation, our so-called *Nebular-IR*. The Nebular-IR allows us to perform optimizations across operators, processing models, and language boundaries. The optimizations range from high-level optimizations on the operator plan level (e.g., placement, ordering, fusion [18]) to low-level optimizations on the instruction level (e.g., branch conversion across operators). One particular challenge for the Nebular-IR is to handle and optimize UDFs. In particular, most data processing systems treat UDFs as black boxes and thus provide only basic optimizations to plans containing them. However, in NES we first analyze UDFs to perform high-level optimizations on the IR (e.g., operator reordering [20]). After that, we fuse operators across UDF-boundaries and generate compact machine code. This allows NES to achieve high code efficiency among different UDFs.

From a management point of view, centrally managing the system in a heterogeneous distributed setup introduces challenges from areas such as data collection, response time, and fault-tolerance. To this end, NES provides a management view with a centralized, homogeneous interface, automatic distribution and parallelization, and means to adaptively detect and react to changes in the environment. Although the management is performed centrally, parts of the system require a decentralized design.

By providing a central management view as well as an intermediate representation in NES, we support a diverse set of data processing workloads specified in different query languages and following different processing models.

3.4.5 C5 - Constant Evolution under Continuous Operation

NebulaStream's goal is to support continuous operations while the topology and user workloads change constantly. A particular challenge originates from a changing topology where new devices join the fog/WSN and existing devices get phased out or change their geo-spatial position. Additionally, the workloads continuously change as users submit, update, or delete queries. Furthermore, to enable time-sensitive processing, nodes must behave dynamically and autonomously during runtime, to capture and react to changes in velocity, volume, and variety. Managing and reacting to changes in a robust way while the system is in continuous operation presents drastic challenges to the software architecture and fabric of an IoT data management system.

To support such a highly dynamic environment in NES, we apply the *Autonomous Processing*, *Dynamic Decisions*, and *Incremental Optimizations* design principles. First, NES equips the compute nodes with all necessary components to autonomously react to a wide range of situations. We enrich the Node-EPs with several alternative routes and different options. As a result, if a node detects changes in velocity, volume, or variety, it reacts dynamically at runtime. To this end, nodes require mechanisms to cope with a highly dynamic environment either locally, by interacting with nodes in the neighborhood, or by reaching out to a global coordinator. The possible design space for these changes includes reduction of the sampling rate, dropping of packages, change in the operator order or algorithm, or rerouting of data streams. Second, each software component in NES is designed to allow for the ever-changing network topology and query workloads and to handle some degree of bounded staleness. We expect that this dynamicity will result in a complete redesign of many components and will require new algorithms and protocols. In particular, we plan to incorporate the actor model [8] to capture the dynamic behavior between moving devices. In this model, each device represents either a client, worker, source, or coordinator actor. Using the actor model, we make sure that each device is always in a valid state and that each device can react to a wide range of events autonomously, e.g., lost connection or coordinator change. We plan to use the actor model for coordination between actors, e.g., sending queries or reacting to node failures. Due to the high message overhead of the actor model, we plan to offload data transfer to a more light-weight mechanism, e.g., ZMQ[1]or RabbitMQ[2]. Third, we apply incremental optimizations such that NES modifies a stateful execution plan of a running query in incremental steps rather than in one large change. With each incoming or modified query as well as with each change in data velocity, volume, or variety, NES converges to the optimal NES-EP. Furthermore, we introduce continuous feedback loops between the NES Coordinator and the NES Node Engines in different layers to enable a central management in a heterogeneous distributed setup. In addition, NES re-optimizes the query execution based on dynamic changes in

[1]https://zeromq.org/
[2]https://www.rabbitmq.com/

the workload and environment in an asynchronous process. The trade-off between a centralized orchestration in a coordinator and decentralized decisions in the nodes remains an open research question for the future.

By defining feedback loops between its components and by performing changes incrementally and autonomously, we attempt to make NES resilient against constantly changing user workloads and network topologies. Running AI/ML/-Data Science algorithms based on data sets and streams produced by sensors in the IoT provides explanation models and prediction capabilities, which in conjunction with actuators, result in a feedback loop between the digital and the physical world. Additionally, programming and reasoning about sensors and actuators in a distributed, diverse setting at the scale of the IoT provides a huge challenge both with respect to ease of use and scalability.

Overall, NebulaStream addresses all challenges of an IoT data management system presented in Section 3.4 by combining existing approaches with new solutions. To this end, NebulaStream's goals are to handle heterogeneous and distributed data sources and formats, to utilize available resources efficiently, to cope with unstable network topologies, and to provide multiple query and processing models. We envision that NES's unique features make it an attractive platform for future IoT application scenarios.

4. STATE-OF-THE-ART SYSTEMS

In this section, we group existing approaches and outline how they address IoT data management challenges.

4.1 Cloud-centric IoT data processing

The first group of approaches relies on the cloud to process IoT data centrally. Mobile Cloud Computing (MCC) outsources data storage and processing from devices to the cloud. In this scenario, a pool of sensors gathers and sends data directly to a cloud infrastructure for further processing [6, 31]. Example applications following this approach are camera surveillance [14, 35], wearable cognitive assistance [15, 33], and smart city monitoring [39, 38]. As soon as data reach the cloud, common SPEs, such as Apache Flink [52, 42, 47] and Apache Pulsar [26, 9] process the incoming streams. Based on this infrastructure, cloud providers offer services to deploy and manage data streams.

The cloud-centric processing of sensor data enables elastic scaling of compute and storage resources once data reach the cloud. However, this neglects the resources provided by sensors and intermediate nodes (**C1**,**C2**). Although these systems offer fault-tolerance and dynamic scaling (addressing **C3**,**C5**) in the cloud, they do not provide them across a unified sensor-fog-cloud environment. In NES, we extend existing work in the area of stream processing to incorporate IoT specific requirements. In particular, we enable cross-paradigm optimization, in-network processing, and hardware-tailored code generation.

4.2 Edge-Aware IoT data processing

With the concept of Mobile-Edge Computing (MEC), cloud providers address the limitations of cloud-centric approaches by implementing *hub devices* to extend their IoT services [48, 5, 32]. Hub devices are placed at the edge of the fog topology and act as local control centers which are close to the sensors. They gather data from attached sensors, perform simple processing steps, and do not require a stable connection to a cloud infrastructure.

Although MEC and MCC improve scalability with respect to the number of sensors (addressing **C1**), they do not focus on efficient resource utilization across heterogeneous devices (**C2**). In particular, hub devices do not enable cooperative processing across the whole topology. Furthermore, these approaches offer fault-tolerance only between hub-devices and the cloud but still require a stable connection between sensors and the hub-device (partially addressing **C3**). Additionally, these approaches do not address dynamic changes in the topology (**C3**).

Ryden et al. [44] introduce a distributed data and resource management framework. They leverage distributed in-situ data and computing resources on edge nodes only for batch processing. Their system supports the combination of dedicated and voluntary resources under a unified infrastructure while ensuring high availability (addressing **C5**, partially addressing **C1**). However, their framework neither exploits hardware heterogeneity for efficient code computation nor supports a multi-programming environment (**C2, C4**). In NES, we support streaming queries in a unified sensor-fog-cloud environment that is able to exploit fog devices and sensors to optimize query execution in a holistic way.

4.3 Fog-aware IoT data processing

Two data processing systems utilize the fog as the underlying infrastructure. O'Keeffe et al. [37] propose Frontier, a distributed and resilient data processing system for fog devices. Frontier aims to handle a large number of sensors and to achieve reliability. To this end, it exploits the processing capability of the fog by distributing queries over a topology (addressing **C1**). It replicates operators to neighboring nodes to recompute intermediate results and to cope with device failures (addressing **C3**). However, Frontier does not address the efficient utilization of heterogeneous devices, diversity in programming environments, and adaptability to the constant evolution of the fog (**C2**,**C4**,**C5**). Finally, it does not consider the exploitation of cloud resources.

Zhitao et al. [48] extend Cisco's Connected Streaming Analytics platform (CSA) for IoT processing. CSA utilizes Cisco network hardware to enable in-network processing (partially addressing **C1**,**C2**). However, CSA does not address potentially unreliable connections, the dynamic evolution of the fog, and provides only an SQL-like interface (**C3**,**C5**,**C4**).

In NES, we build on top of these approaches and combine the possible compute and storage capacities of the fog and the cloud. Besides Frontier and CSA, additional research has been conducted on individual challenges in fog computing, which we will leverage in NES. Janssen et al. [22] propose operator placement techniques to partition queries across a fog topology (addressing **C1**). Park et al. [41] exploit special capabilities of IoT hardware to improve efficiency and security (addressing **C2**). Kang et al. [23] and Grulich et al. [17] propose solutions to partition the inference of deep neural networks across fog topologies to improve scalability (addressing **C1**).

4.4 Data Processing in Sensor Networks

Sensor networks (SNs) target a particular sub-area of the IoT [30, 53]. In particular, these systems focus on distributed processing in a wireless sensor network [54]. A ma-

jor goal is resilience to intermittent and changing network connectivities. To this end, sensor nodes form a network to transfer sensor values through multiple hops to a root node and perform in-network data processing. Approaches in this area tackle efficiency (addressing **C2**) by optimizing the computation for battery lifetimes and enable filtering and aggregation queries over sensor data [30]. Moreover, they provide support for a dynamic execution environment (addressing **C5**). However, these approaches do not support more complex and general workloads, which combine multiple queries, languages, and algebras (**C4**). In addition, they do not provide strong fault-tolerance and correctness guarantees (**C3**).

In NES, we leverage concepts from sensor networks and integrate them seamlessly across the Sensor, Fog, and Cloud Layers, resulting in a unified environment.

5. CONCLUSION

In this paper, we introduced NebulaStream, a general-purpose, end-to-end data management system for the IoT. We showed that current systems are not yet ready for the upcoming challenges of the IoT era. We highlighted the system design of the NebulaStream platform and its design principles. The goal of our envisioned design is to handle the heterogeneity, unreliability, and elasticity of a unified sensor-fog-cloud environment. Furthermore, we revealed upcoming research challenges and outlined possible solutions. Finally, we presented first results that motivate the need of a new system design for upcoming IoT applications. With our NebulaStream Platform, we aim to enable emerging IoT applications in different domains.

6. ACKNOWLEDGMENTS

This work was funded by the EU projects E2Data (780245), DFG Priority Program "Scalable Data Management for Future Hardware" (MA4662-5), FogGuru (Horizon 2020 under Marie Skłodowska-Curie grant agreement No 765452), the German Ministry for Education and Research as BBDC II (01IS18025A), and by the German Federal Ministry for Economic Affairs and Energy as Project ExDra (01MD19002B). This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein. Bonaventura Del Monte is partially funded by the German Ministry for Education and Research as Software Campus 2.0 (01IS17052). We thank Julius Hülsmann for his support with the experiments on decentralized joins and Vianney de Cibeins for his support with the experiments on data reduction techniques. Furthermore, we thank Eleni Tzirita Zacharatou and Xenofon Chatziliadis for the valuable input and discussions.

7. REFERENCES

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, 2005.

[2] A. Alexandrov et al. The stratosphere platform for big data analytics. *VLDB Journal*, 2014.

[3] A. Alexandrov, A. Kunft, A. Katsifodimos, F. Schüler, L. Thamsen, O. Kao, T. Herb, and V. Markl. Implicit parallelism through deep language embedding. *SIGMOD Rec.*, 2016.

[4] G. Alonso, C. Binnig, I. Pandis, K. Salem, J. Skrzypczak, R. Stutsman, L. Thostrup, T. Wang, Z. Wang, and T. Ziegler. DPI: the data processing interface for modern networks. In *CIDR*, 2019.

[5] Amazon. Amazon aws greengrass, 2017. Retrieved December 15, 2019, from https://aws.amazon.com/greengrass.

[6] Amazon. Aws iot analytics, 2018. Retrieved December 15, 2019, from https://aws.amazon.com/iot-analytics.

[7] M. Balazinska, J. Hwang, and M. A. Shah. Fault tolerance and high availability in data stream management systems. In *Encyclopedia of Database Systems, Second Edition*. 2018.

[8] P. A. Bernstein, S. Burckhardt, S. Bykov, N. Crooks, J. M. Faleiro, G. Kliot, A. Kumbhare, M. R. Rahman, V. Shah, A. Szekeres, and J. Thelin. Geo-distribution of actor-based services. *PACMPL*, (OOPSLA), 2017.

[9] J. Bock. Solving the challenges of iot analytics. Retrieved December 15, 2019, from https://streaml.io/blog/solving-the-challenges-of-iot-analytics.

[10] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Mobile Cloud Computing (MCC)*, 2012.

[11] P. Carbone, S. Ewen, G. Fóra, et al. State management in apache flink: Consistent stateful distributed stream processing. In *VLDB*, 2017.

[12] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *IEEE international parallel and distributed processing symposium workshops (IPDPSW)*, 2016.

[13] B. Del Monte. Efficient migration of very large distributed state for scalable stream processing. In *Proceedings of the VLDB PhD Workshop*, 2017.

[14] Google. Nest cam, 2017. Retrieved December 15, 2019, from https://nest.com/cameras.

[15] Google. Glass enterprise edition 2, 2019. Retrieved December 15, 2019, from https://www.google.com/glass.

[16] J. Grier. Extending the yahoo! streaming benchmark, 2016. Retrieved December 15, 2019, from http://data-artisans.com/extending-the-yahoo-streamingbenchmark.

[17] P. M. Grulich and F. Nawab. Collaborative edge and cloud neural networks for real-time video processing. In *VLDB*, 2018.

[18] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. In *ACM Computing Surveys (CSUR)*, 2014.

[19] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *SIGCOMM*, 2013.

[20] F. Hueske, M. Peters, M. J. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening

the black boxes in data flow optimization. In *VLDB*, 2012.

[21] M. Hung. Leading the iot, gartner insights on how to lead in a connected world. *Gartner Research*, 2017.

[22] G. Janßen, I. Verbitskiy, T. Renner, and L. Thamsen. Scheduling stream processing tasks on geo-distributed heterogeneous resources. In *IEEE International Conference on Big Data (Big Data)*, 2018.

[23] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *ACM SIGARCH Computer Architecture News*, 2017.

[24] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. Benchmarking distributed stream data processing systems. In *International Conference on Data Engineering, ICDE*, 2018.

[25] J. Karimov, T. Rabl, and V. Markl. Astream: Ad-hoc shared stream processing. In *SIGMOD*, 2019.

[26] D. Kjerrumgaard. Using apache pulsar to provide real-time iot analytics on the edge. Retrieved December 15, 2019, from https://de.slideshare.net/streamlio/streamlio-and-iot-analytics-with-apache-pulsar.

[27] L. Kroll, K. Segeljakt, P. Carbone, C. Schulte, and S. Haridi. Arc: an ir for batch and stream programming. In *International Symposium on Database Programming Languages (SIGPLAN)*, 2019.

[28] A. Kunft, A. Katsifodimos, S. Schelter, S. Breß, T. Rabl, and V. Markl. An intermediate representation for optimizing machine learning pipelines. In *VLDB*, 2019.

[29] A. Lerner, R. Hussein, and P. Cudré-Mauroux. The case for network accelerated query processing. In *CIDR*, 2019.

[30] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. In *ACM Transactions on database systems (TODS)*. ACM, 2005.

[31] Microsoft. Azure iot hub, 2016. Retrieved December 15, 2019, from https://azure.microsoft.com/en-us/services/iot-hub/.

[32] Microsoft. Microsoft azure iot edge, 2017. Retrieved December 15, 2019, from https://azure.microsoft.com/en-us/services/iot-edge.

[33] Microsoft. Hololens 2, 2019. Retrieved December 15, 2019, from https://www.microsoft.com/en-us/hololens.

[34] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac. Internet of things: Vision, applications and research challenges. In *Ad Hoc Networks*, 2012.

[35] Netatmo. Netatmo cam, 2017. Retrieved December 15, 2019, from https://www.netatmo.com.

[36] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *VLDB*, 2011.

[37] D. O'Keeffe, T. Salonidis, and P. Pietzuch. Frontier: Resilient edge processing for the internet of things. In *VLDB*, 2018.

[38] OpenFog Consortium. Smart cities scenario (3.3), 2017. Retrieved December 15, 2019, from https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf.

[39] OpenFog Consortium. Visual security and surveillance scenario (3.2), 2017. Retrieved December 15, 2019, from https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf.

[40] S. Palkar, J. J. Thomas, A. Shanbhag, D. Narayanan, H. Pirk, M. Schwarzkopf, S. Amarasinghe, M. Zaharia, and S. InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[41] H. Park, S. Zhai, L. Lu, and F. X. Lin. Streambox-tz: A secure iot analytics engine at the edge. In *Computing Research Repository (CoRR)*, 2018.

[42] J. Piasecki. 7 reasons to use real-time data streaming and flink for your iot project. Retrieved December 15, 2019, from https://freeportmetrics.com/blog/7-reasons-to-use-real-time-data-streaming-and-flink-for-your-iot-project/.

[43] D. Reinsel, J. Gantz, and J. Rydning. Data age 2025: The digitization of the world from edge to core., 2018. Retrieved December 15, 2019, from https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf.

[44] M. Ryden, K. Oh, A. Chandra, and J. Weissman. Nebula: Distributed edge cloud for data intensive computing. In *ICDE*, 2014.

[45] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag. Streams and tables: Two sides of the same coin. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, BIRTE '18, 2018.

[46] T. Schneider. Analyzing 1.1 billion nyc taxi and uber trips, with a vengeance, 2015. Retrieved December 15, 2019, from http://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/.

[47] M. Sfikas. Smart systems iot use case with open source kafka, flink & cratedb. Retrieved December 15, 2019, from https://www.ververica.com/blog/smart-systems-iot-use-case-open-source-kafka-flink-cratedb.

[48] Z. Shen, V. Kumaran, M. J. Franklin, S. Krishnamurthy, A. Bhat, M. Kumar, R. Lerche, and K. Macpherson. Csa: Streaming engine for internet of things. In *IEEE Data Eng. Bull.*, 2015.

[49] J. Traub, S. Breß, T. Rabl, A. Katsifodimos, and V. Markl. Optimized on-demand data streaming from sensor nodes. In *SoCC*, 2017.

[50] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In *EDBT*, 2019.

[51] J. Traub, J. Hülsmann, S. Breß, T. Rabl, and V. Markl. SENSE: Scalable data acquisition from distributed sensors with guaranteed time coherence. 2019. arXiv 1912.04648; https://arxiv.org/abs/1912.04648.

[52] S. Yang. Iot stream processing and analytics in the fog. *IEEE Communications Magazine*, 55, Aug 2017.

[53] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. In *SIGMOD*. ACM, 2002.

[54] J. Yick, B. Mukherjee, and D. Ghosal. Wireless sensor network survey. In *Computer networks*, 2008.

[55] M. Zaharia, R. S. Xin, P. Wendell, et al. Apache spark: a unified engine for big data processing. In *Communications of the ACM*, 2016.

[56] S. Zeuch, B. Del Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl. Analyz-

ing efficient stream processing on modern hardware. In *VLDB*, 2019.

[57] S. Zeuch and J. Freytag. QTM: modelling query execution with tasks. In *ADMS*, 2014.