Demand-based Sensor Data Gathering with Multi-Query Optimization

Julius Hülsmann
Technische Universität Berlin

Jonas Traub
Technische Universität Berlin

Volker Markl
Technische Universität Berlin

ABSTRACT

In the Internet of Things, billions of sensors provide data streams to applications. The data are predominately acquired from devices with constrained computational capabilities, often serving multiple queries simultaneously. Sensor nodes, are typically oblivious to the specific needs of applications. The potential requirements of diverse applications force them to push data at a higher rate than required by a specific, currently running application. That is suboptimal due to 1. constraints in the network bandwidth, 2. expenses for transmissions, and 3. limited computational power. However, decreasing data gathering frequency may reduce the applications' accuracy. In this paper, we demonstrate a technique for minimizing the number of network transmissions while maintaining the desired accuracy. The presented algorithm for read- and transmission-sharing among queries goes hand-in-hand with state-of-the-art machine learning techniques for adaptive sampling. We 1. implement the technique and deploy it on a sensor node, 2. replay sensor-data from two real-world scenarios, 3. provide an interface for submitting custom queries, and 4. present an interactive dashboard. Here, visitors observe live statistics on the read- and transmission savings achieved in real-world use-cases. The dashboard also visualizes optimizations currently performed by the read scheduling procedure and hence conveys real-time insights and a deep understanding of the presented algorithm.

PVLDB Reference Format:
Julius Hülsmann, Jonas Traub, Volker Markl. Demand-based Sensor Data Gathering with Multi-Query Optimization. *PVLDB*, 12(xxx): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

The continued growth of the Internet of Things (IoT) [12] enables the development of disruptive applications in a variety of different domains [1, 3]. Such applications share a need for input data, typically provided as data streams and gathered from distributed sensor nodes. Each sensor node possibly provides data points required for several different queries. In this demonstration, we present two different real-world use-cases, which share the same input data: **Application 1** learns driving profiles of vehicles and assesses the driver's aggressiveness. **Application 2** tracks the number of vehicles in a specific area. Both applications process the velocity of public transport vehicles in Berlin[14].

The two applications differ in their data-demand, i.e., the minimum number of data points required for answering involved queries with the desired precision. Especially in Application 1, the velocity of observed vehicles determines the data demand; it requires a high sampling frequency for vehicles moving at a high pace and fewer data records from slow vehicles. Current stream processing engines (SPEs) [2, 9] are demand-oblivious; they gather, transmit, and process as much data as possible. Running multiple queries for the same sensor opens up for the possibility to share reads and transmissions among them, e.g., by serving multiple requests with one single sensor read. This opportunity is regularly left unexploited when gathering data from distributed sensor nodes. The resulting unnecessary data transmissions impose scalability issues or unnecessary charges for network traffic and system scale-out [10]. On the other hand, delayed or infrequent input can harm the output precision.

Designing a demand-based processing pipeline that adheres to requirement specifications from data consumers is challenging. It warrants (i) a flexible way of specifying a query's data-demand, which also (ii) has to allow for sharing reads among queries.

The *specification of the data-demand* has to be flexible enough to embrace diverse data-demands. For instance, in Application 1, the data-demand depends on previously gathered data. This dependency has to be reflected in the demand-specification. We express data-demands as stateful user-defined functions, which continuously determine when to schedule sensor reads. *Sharing reads and transmissions* among queries builds on top of demand-based scheduling. If Applications 1 and 2 run concurrently, the system needs to be able to determine whether sharing a sensor read among the two associated queries complies with the queries' data-demands. To efficiently share sensor reads and transmissions among queries, the data-consumer can specify tolerated deviations from the demand. We then use this information to compute a suitable time to perform a single read for multiple queries, whenever in accordance with the data-demand.

We demonstrate a demand-based technique for sensor data gathering, that shares sensor reads and data transmissions

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

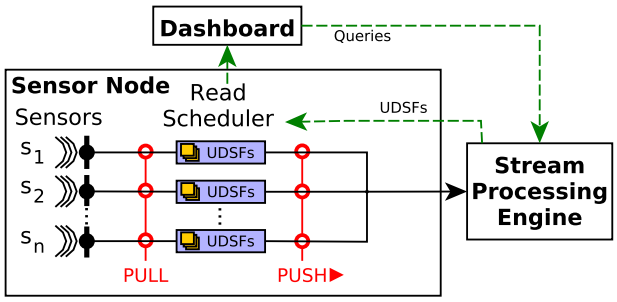

Figure 1: An SPE receives queries from the user and conveys their data-demand to sensor nodes. The nodes schedule sensor reads and share them among queries. The dashboard provides live insights into the internals of the Read Scheduler.

among queries [10]. Therefore, we provide an interactive demonstration with configurable queries and associated data-demands that we execute on a sensor device.

We implemented the presented technique as a feature of NebulaStream [14], and offer real-time insights into the optimization process. We use an open-source framework for replaying data from two real-world datasets on the sensor node [5]: The first dataset, on which we execute queries for the introduced example applications, contains telemetry data of vehicles. The data stems from an integrated the public transport system of Berlin [14]. We replay the velocity of different vehilces (i.e. taxis, buses, subways). The second dataset is provided by the DEBS'13 Grand Challenge and contains the speed of a football [7].

The audience can define custom queries on the datasets and monitor the read- and transmission-savings our algorithm achieves in a realistic scenario. Our dashboard deepens the understanding of the proposed techniques by providing live insights into the performed optimizations through live visualizations.

To summarize, we make the following contributions:

1. We implement a technique for gathering data from sensor nodes and deploy it on a Raspberry Pi. Our technique optimizes the data gathering process based on the data-demands of concurrently running queries.

2. We provide an interactive dashboard for live monitoring of (i) the performed optimizations and (ii) the savings in data reads and transmission (Figure 4).

3. Visitors of our demonstration can configure custom queries and data-demands to observe the algorithm react to changing data requirements and observe the solution's performance in realistic scenarios.

The remainder of this paper is structured as follows: In Section 2, we discuss the demonstrated techniques. In Section 3, we describe the setup of our demonstration and the visualizations presented on our dashboard. Finally, we discuss related work in Section 4 and conclude in Section 5.

2. DEMAND-BASED DATA GATHERING

In this section, we first provide an overview of our solution in Section 2.1, then describe demand specification in Section 2.2, and multi-query optimization in Section 2.3.

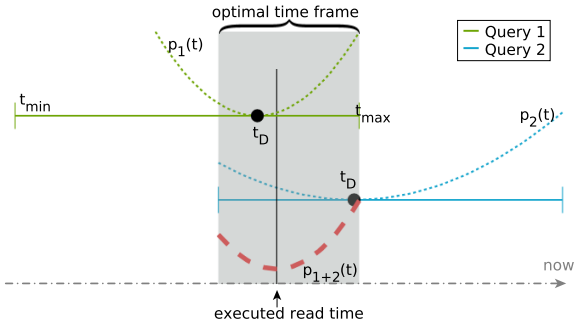

Figure 2: Read time suggestion and read-fusion of two concurrent queries (green and blue).

2.1 Solution Overview

Figure 1 shows the high-level architecture of our solution. The user perspective remains unchanged compared to traditional stream analysis systems. The user submits queries to a stream analysis cluster, which is represented by the stream processing engine (SPE) in Figure 1. The SPE forwards the query to the relevant sensor nodes, and reports results back to the user. While traditional SPEs are push-based and have no control over the incoming data streams, our approach combines push-based data processing with pull-based sensor reads. The read-scheduler on the sensor node pulls (i.e., reads) data from sensors based on the overall data-demand posted by all running queries. Then, the scheduler streams (i.e., pushes) the data asynchronously through the processing pipeline, allowing for low latency processing.

When submitting a query, the data-consumer specifies the data-demand. The next two sections describe the demand specification and multi-query optimization in detail.

2.2 Demand Specification

Our proposed approach to data gathering consists of two core components: 1. Scheduling the optimal next read time for a query depending on its data-demand, and 2. merging similar read times to share reads and transmissions. Therefore, we extend queries by the following information:

(i) A sampling algorithm $t_D(\cdot)$ specifies a query's data-demand by proposing the *desired read timestamp*, based on the last sensor read $\langle time, value \rangle$ as well as the last desired read time. Users can specify $t_D(\cdot)$ as an analytical expression (i.e., user-defined function), which is very powerful and flexible. Alternatively, users can pick from existing adaptive sampling algorithms such as Adam [11] and FAST [4]. These algorithms adapt to the volatility in the observed sensor values and adjust sampling frequencies to capture the underlying time series with a tolerated error, while minimizing the number of required sensor readings.

(ii) As the data-demand differs among queries, efficient read- and transmission-reduction requires slight deviations from the desired read time. The tolerance to such deviations depends on the application. Hence, for each query, bounds $t_{min}(\cdot), t_{max}(\cdot)$ around the desired read time indicate the acceptable time interval for the next read. A penalty function $p(\cdot)$ is used to compare the negative impact of deviations from the desired read time. The data consumer specifies the bounds as analytic expressions, based on the next desired read time, and the last sensor read $\langle time, value \rangle$. The penalty is required to be a convex function, centered at the

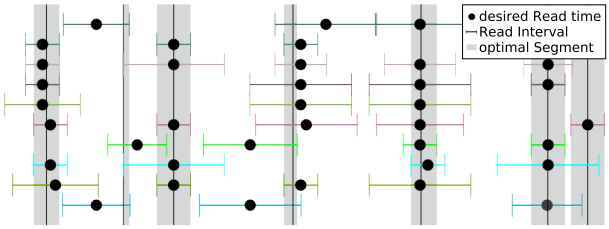

Figure 3: Plot of optimal time frames for sensor reads and selected read times with ten concurrent queries on the public transport dataset.

next desired read time. The so-called *user-defined sampling function* (UDSF) condenses *(i)* and *(ii)* into one function:

$$\langle time, value \rangle \mapsto \langle t_{min},\ t_D,\ t_{max},\ p(t) \rangle$$

After each sensor read, the function is called with the *time* and the *value* of the reading as parameters. The UDSF, including the second order function $p(t)$, is specified by the user and returns the instructions for the next sensor reading.

Examples: We run queries for Applications 1 and 2 from the introduction on the data replayed from the Berlin public transport dataset [14]. We illustrate the UDSFs used in these scenarios hereafter. First, consider Application 2, which tracks the number of vehicles in a specific area. It only issues a sensor read if a vehicle is potentially close to switching areas. In order to account for changes in velocity, this translates to taking at least one measurement per minute, or if the distance between the vehicle and the boundaries of the area will be crossed when traveling at a constant velocity. Reading up to two seconds early is fine, and reading one second late is allowed. The UDSF is defined on the vehicle's (positive) velocity v [m/s] and current position p:

$$\mathfrak{U}_2 := \langle t_D - 2\text{s},\ t + \min(60s, \frac{\text{dist}(p, area)}{v}),\ t_D + 1\text{s},\ 20 \cdot t^2 \rangle$$

The UDSF for the driving behavior profiling application (1) is much more strict and requires data at least once a second or every 20 meters. The penalty of deviating from the desired read time increases with the vehicle's velocity.

$$\mathfrak{U}_1 := \langle t_D - 0.1\text{s},\ t + \min(1s, \frac{20m}{v}),\ t_D + 0.1\text{s},\ (v+1) \cdot t^2 \rangle$$

It is possible to disable read-sharing altogether by enforcing exact reads at the proposed time. We demonstrate this behavior with queries on the DEBS'13 football dataset by defining the following UDSFS: $\mathfrak{U}_a$ uses the adaptive sampling algorithm AdaM and $\mathfrak{U}_b$ samples at the maximum possible frequency.

$$\mathfrak{U}_a := \langle t_D, AdaM(time, value), t_D, 0 \rangle,\ \mathfrak{U}_b := \langle t_D, t_D^{last} t_D, 0 \rangle$$

Neither $\mathfrak{U}_a$ nor $\mathfrak{U}_b$ permit deviations from desired read times.

2.3 Multi Query Optimization

Figure 2 illustrates the scheduling of a sensor read based on the data-demands of two queries. Both queries have submitted t_{min}, t_D, t_{max}, and $p(t)$ to request the next sensor reading as described above. The scheduler now determines the optimal time frame for the next sensor read (highlighted gray). Reading within this time frame allows for sharing a single sensor read and transmission to satisfy the-demand of

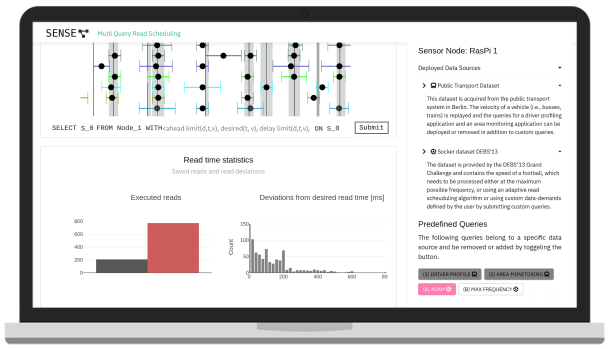

Figure 4: A view into the sensor-dashboard, running the driver profile and area monitoring applications.

both queries. To determine the best read time within the marked time frame, we sum the convex penalty functions $p(t)$ and find the minimum of the resulting sum. Reading at the time of that minimum implies the smallest penalty. We represent the selected read time by a vertical black line. Once we performed the sensor read, we use the read time and the obtained sensor value to request the next desired read (t_{min}, t_D, t_{max}, and $p(t)$) from the UDSFs of the two currently active queries. We refer the reader to our full paper [10] for a thorough specification of UDSF and our scheduling algorithm which goes beyond the summary provided in this section.

Figure 3 shows the scheduler in action for a larger number of queries. Due to the limited space, we do not show the penalty function in this figure. The evaluated UDSFs contribute to the cumulative penalty function in the selected fragments, which is minimized by the algorithm.

3. DEMONSTRATION

To demonstrate our solution, we integrated the read scheduler introduced in Section 2.3 into NebulaStream [14] to gather sensor values from a Raspberry Pi. On the sensor, we use an open-source framework for replaying real-world datasets [5]. We provide a dashboard for live monitoring of the performed optimization and savings in data reads and transmissions. Attendees interact with the algorithm by deploying the queries associated with the real-world examples on the datasets and submitting additional custom queries with associated UDSFs. They gather real-time insights into the performed optimization and observe the transmission achieved savings.

3.1 Setup

We selected a Raspberry Pi 3 Model B single-board computer due to its cost-effectiveness and widespread use in sensor-based research as our sensor node. We equipped the device with the real-time kernel patch PREEMT-RT, allowing for more precise wakeup times for read-execution, scheduled according to the Pi's steady system clock. It is worth noting that our framework operates cross-platform: It runs on POSIX compliant operating systems, GNU/Linux and BSD variants and is agnostic to the underlying hardware architecture.

3.2 Interactive Dashboard

The dashboard consists of three core components: 1. The configuration panel, allowing to provision the Raspberry

Figure 5: Executed reads and deviations from t_D.

Pi with queries, 2. a visualization of the read-sharing, and 3. statistics on the performed reads and transmissions.

Configuration Panel: Attendees submit queries with custom data-demand specifications provided as UDSFs, that are applied immediately. For ease-of-use, a set of pre-configured queries allows us to present various aspects of the solution in detail: Attendees learn about the fundamentals of read-time-suggestion when we present our one-query configuration. We explain read-fusion with the second default configuration, which launches two queries simultaneously. Figure 2 shows a screenshot of this configuration. Deployable queries for the real world examples on the football and public transport datasets constitute realistic data-gathering environments.

Read Sharing Visualization: The read-sharing visualization consists of a continuously updated timeline. It combines the information conveyed in Figures 2 and 3 by visualizing (i) currently active UDSF (namely, the time interval, penalty function, and desired read time) for each query, (ii) the optimal time frame for the next read (shaded area), and (iii) the executed read time (as a vertical line spanning all queries). We can switch off the plotting of penalty functions for larger numbers of queries to avoid clutter.

Read and Traffic Statistics: Read-fusion saves sensor reads and data transmissions but requires tolerances in the desired read times to enable read-fusion. We visualize this tradeoff in the *Read time statistics* panel in Figure 5. On the left-hand side, we contrast the executed number of reads to the amount required when read-fusion was disabled. On the right-hand side, a histogram shows the deviations from the desired read time t_D for past reads.

4. RELATED WORK

While the problem of data-oblivious sampling is studied in the literature, our solution adds more flexibility for demand-based data gathering and addresses multi-query optimization. TinyDB [6] introduces the concept of *acquisitional query processing* to combine database operators and sensor reads in joint pipeline. Our work extends this approach with the ability to run more flexible, non-peroidic, and stateful sampling logic such as adaptive sampling functions [4, 11]. Xiang et al. [13] eliminate read redundancies by scheduling reads at the greatest common divisor of all sampling rates. Read-fusion, exactly adhering to scheduled reads, is a special case of our solution, while we also allow exploiting read time tolerances. In contrast to the approach by Tavakoli et al. [8], UDSFs provide the flexibility to perform non-periodic reads, to specify dynamic read tolerances, and to weight read-times within the acceptable tolerance. In summary, we provide a combined framework for Demand-based Data Gathering and Multi-Query Optimization. Hereby, separation of the specification for read-suggestion and read-fusion allows us to correctly address a query's data-demand.

5. CONCLUSION

We demonstrate a solution for demand-based data gathering from sensor nodes with multi-query optimization. Our solution reduces the number of sensor reads and the amount of transferred data. The demonstration shows the flexibility and the ease-of-use of the proposed approach. Queries express their data-demand as user-defined sampling functions (UDSFs) that enable sharing of sensor reads and data traffic among queries. While the complexity of demand-based read-scheduling is transparent to the user, this demonstration helps understanding the internals and allows for composing user-defined sampling functions that capture the specific data-demands of queries.

6. REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[2] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Big Data Bulletin*, 36(4), 2015.

[3] L. Da Xu, W. He, and S. Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.

[4] L. Fan and L. Xiong. An adaptive approach to real-time aggregate monitoring with differential privacy. *IEEE TKDE*, 26(9), 2014.

[5] D. Giouroukis, J. Hülsmann, J. von Bleichert, M. Geldenhuys, T. Stullich, F. Gutierrez, J. Traub, K. Beedkar, and V. Markl. Resense: Transparent record and replay of sensor data in the Internet of Things. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2019.

[6] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *TODS*, 30(1), 2005.

[7] C. Mutschler, H. Ziekow, and Z. Jerzak. The debs 2013 grand challenge. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 289–294, 2013.

[8] A. Tavakoli, A. Kansal, and S. Nath. On-line sensing task optimization for shared sensors. IPSN, 2010.

[9] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@twitter. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 147–156, 2014.

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

[11] D. Trihinas, G. Pallis, and M. D. Dikaiakos. AdaM: An adaptive monitoring framework for sampling and filtering on IoT devices. IEEE Big Data, 2015.

[12] R. van der Meulen. Gartner says 6.4 billion connected things will be in use in 2016, up 30 percent from 2015. 2015.

[13] S. Xiang, H. B. Lim, K.-L. Tan, and Y. Zhou. Two-tier multiple query optimization for sensor networks. ICDCS, 2007.

[14] S. Zeuch, A. Chaudhary, B. Del 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. In *Conference on Innovative Data Systems Research (CIDR)*, 2019.