Optimized On-Demand Data Streaming from Sensor Nodes

Jonas Traub[1] Sebastian Breß[1,2] Tilmann Rabl[1,2] Asterios Katsifodimos[3] Volker Markl[1,2]

[1]*Technische Universität Berlin* [2]*German Research Center for Artificial Intelligence (DFKI)* [3]*SAP Innovation Center*

jonas.traub@tu-berlin.de sebastian.bress@dfki.de rabl@tu-berlin.de asterios.katsifodimos@sap.com volker.markl@tu-berlin.de

ABSTRACT

Real-time sensor data enables diverse applications such as smart metering, traffic monitoring, and sport analysis. In the Internet of Things, billions of sensor nodes form a *sensor cloud* and offer data streams to analysis systems. However, it is impossible to transfer all available data with maximal frequencies to all applications. Therefore, *we need to tailor data streams to the demand of applications.*

We contribute a technique that optimizes communication costs while maintaining the desired accuracy. Our technique schedules reads across huge amounts of sensors based on the data-demands of a huge amount of concurrent queries. We introduce user-defined sampling functions that define the data-demand of queries and facilitate various adaptive sampling techniques, which decrease the amount of transferred data. Moreover, we share sensor reads and data transfers among queries. Our experiments with real-world data show that our approach saves up to 87% in data transmissions.

CCS CONCEPTS

• **Computer systems organization** → *Sensor networks*; *Real-time system architecture*; • **Information systems** → *Data streams*;

KEYWORDS

Sensor Data, Real-Time Analysis, On-Demand Streaming, Oversampling, User-defined sampling, Adaptive Sampling, Sensor Sharing

ACM Reference Format:
Jonas Traub, Sebastian Breß, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 2017. Optimized On-Demand Data Streaming from Sensor Nodes. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 12 pages.
https://doi.org/10.1145/3127479.3131621

1 INTRODUCTION

Billions of devices are equipped with sensors to supply data analysis applications with real-time data [43]. The resulting vast amount of data streams causes heavy network utilization and scalability challenges, which incur increased financial costs. Currenly, sensor data analysis follows a monolith architecture with a tight coupling of applications to sensors. However, the Internet of Things (IoT) works as a *sensor cloud* which is shared among applications. This requires us to break away from monolith architectures and to introduce a new architecture which decouples sensor management from introducing new applications.

Data requirements differ significantly among use-cases. For example, outlier detection requires high sampling frequencies and has low selectivity in local filters at the sensor node. The opposite is true for monitoring a long term trend in time series, which has a low sampling frequency and does not apply local filters. Real-time analysis engines (e.g., Apache Flink [1] or Storm [39]) require data at high frequencies to serve all possible use-cases. This is suboptimal because it forces to read and transfer sensor values beyond the data demand of queries. We call this *oversampling*. More formally, the *data demand* of a query is the minimum number of data points which allows for answering the query with the desired precision. *Oversampling* is reading or transferring additional data points that are not required to achieve the desired result precision.

The massive growth in the amount of sensors is a game changer, which makes *oversampling* a critical problem: as the number of available sources increases rapidly, it is unaffordable to process all available inputs with maximal frequencies. Thus, we need to trade-off sampling rates against system scale-out costs and data transfer charges.

It is challenging to prevent oversampling. Periodic sampling reads and transfers data with a fixed frequency. This is insufficient due to *missing adaptivity*: adaptive sampling techniques dynamically adjust sampling rates depending on the variance within recent samples [17, 19, 41]. With adaptive sampling, we retrieve detailed data (high sampling rate) from sensors which experience anomalies. However, most sensors do not experience anomalies at the moment and reduce their sampling rates. Thus, at any time, we process high frequency data from a few sensors but we reduce sensor reads, data transmissions, and processing effort for the majority of sensors.

Adaptive sampling techniques provide good approximations of time series with significantly reduced average sampling rates compared to periodic sampling. However, adaptive sampling is impractical for other use-cases such as outlier or failure detection. There is no *one-for-all* sampling technique, which at the same time serves all queries and prevents oversampling. The naive approach to set up a smart sampling technique for each query independently is not satisfying either. It might avoid oversampling for one query, but it disregards commonalities between multiple queries, which from a global point of view, again causes *oversampling* and *redundant data transmissions*. Current real-time analysis platforms do not take control of the production of their input streams [7, 39].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SoCC '17, September 24–27, 2017, Santa Clara, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3131621

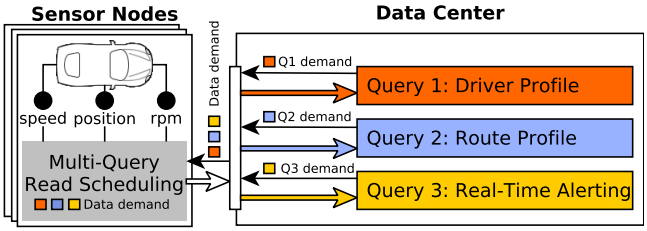
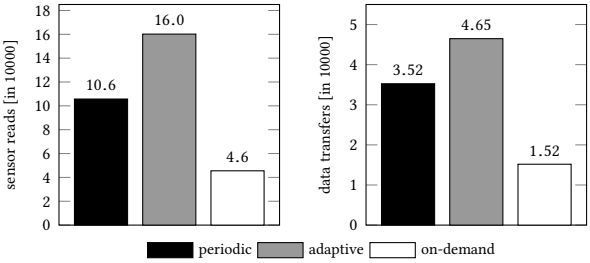


Figure 1: Multi-query read scheduling provides tailored data streams based on the data demand of queries.



(a) Number of sensor reads. **(b) Transferred tuples.**

Figure 2: Sensor reads and transferred tuples for our introductory use-case on Formula 1 data.

Instead, they rely on techniques such as load shedding [37] and back pressure handling, to avoid system crashes when data rates increase. Both techniques run centrally, after transferring the data from sensor nodes to a stream analysis system. Thus, they neither prevent oversampling nor redundant data transmissions.

Common sensor networks such as TinyDB [26] and Cougar [14] compile queries locally at a base station and then disseminate them to sensor nodes. Thereby, they focus on the optimization of a *single query*. This paper complements existing sensor networks by enabling the *sharing of sensor reads and traffic costs among queries*.

In this paper, we introduce *on-demand streaming from sensor nodes*. While we make all data accessible, *ideally, the amount of read and transferred data should solely depend on the demand of executed queries instead of the amount of theoretically available data.*

Our solution consists of two components:

First, we solve the *oversampling problem* using *user-defined sampling functions (UDSFs)*. UDSFs allow for publishing the data demand of queries to data gathering components, which can then provide well orchestrated data streams. UDSFs are highly flexible, easy to implement, and keep the complexity of multi-query optimization transparent to the user.

Second, we solve the *redundant transmission problem* with an algorithm for *multi-query read scheduling*, which is executed at the sensor nodes. Our algorithm executes the minimum possible number of sensor reads only. Therefore, it shares sensor reads and traffic among queries and optimizes the times when sensor reads are performed. Summarizing, our contributions are as follows:

(1) We introduce *user-defined sampling functions (UDSFs)* to overcome the *missing adaptivity* of periodic sampling and to avoid *oversampling*.
(2) We contribute a *multi-query read scheduling algorithm*, which enables frequent read and traffic sharing among queries to avoid *redundant data transmissions*.
(3) We further *optimize read times* based on given read time preferences while still executing only the *minimum number of reads* in total.
(4) We experimentally validate our approach and show its effectiveness in a practical setting.

UDSFs and *read scheduling* aid various use-cases such as traffic monitoring (Section 2), sport analysis [31], and smart metering [4]. Our evaluation shows that our approach reduces data transmissions and sensor reads by up to 87% and scales to hundreds of queries.

2 A MOTIVATING EXAMPLE

We show our solution with an example in Figure 1. We use floating car data to provide alerts to drivers ahead of dangerous locations, which often cause heavy braking (e.g., tight curves or animal crossings). Similar assistance systems use floating car data for green light optimal speed control [32] and online traffic estimation [34].

Three queries are required in our example: Query 1 retrieves data to train a driver profile with a machine learning technique. Query 2 retrieves data to train a route profile. Query 3 combines route and driver profiles with current telemetry data to detect exceptional situations, which then leads to alerts.

Each query has a different data demand: Query 1 observes the aggressiveness of drivers (intensity of breaking and acceleration). Therefore, it adaptively increases sampling rates when accelerating or braking. Query 2 requires a sample at least every 20 meters to profile the road and, therefore, computes the next sensor read time as $t = \frac{20m}{\text{current speed}}$. Query 3 requires a sample at least every 0.3s.

We simulate our example with telemetry data from Formula 1 cars. Therefore, we replay sensor data from the fastest qualifying laps of 32 Formula 1 races in 2015 and 2016 with a 30Hz sampling rate. We utilize tolerances in sensor read times: Query 1 uses adaptive sampling with ±0.2s read time tolerance. Query 2 and 3 enforce minimum sampling rates, but allow higher rates.

Each query defines its data demand and read time tolerances in a UDSF. UDSFs empower domain experts to specify the data demand without specifying details of the query execution. For example, we use domain knowledge to determine proper tolerance intervals for read times. We found that ±0.2s read time tolerance in Query 1 provide the best trade-off between result accuracy and savings achieved through sensor read sharing.

We show the number of sensor reads and data transfers in Figure 2. *Periodic* sampling falls back to the highest sampling rate which is requested by any UDSF at any time. This results in more than 100 thousand sensor reads. *On-Demand* scheduling saves 57% in sensor reads compared to *periodic* sampling because it can adapt sampling rates at runtime. *Adaptive* sampling can reduce sampling rates most of the time. However, when executing queries independently, adaptivity does not make up for the missed opportunity to share sensor reads among the queries. Respectively, *On-Demand* scheduling saves 72% of the sensor reads compared to executing queries independently.

We combine values from three sensors (speed, position, and rpm) in each tuple. Thus, the number of transferred tuples is about 1/3 of the number of sensor reads. Additionally, adaptive sampling avoids transfers with adaptive filtering. We discuss adaptive filtering in detail in Section 5.5.

The reduction in data transmissions would cut charges for mobile network usage when monitoring a fleet of cars. Additionally, the reduced inbound traffic at a central analysis cluster prevents scale-out fees of cloud providers.

3 BACKGROUND

Before we discuss *UDSFs* and our *multi-query read scheduling algorithm*, we provide an overview of sensor data transfer, adaptive sampling, and usage scenarios.

3.1 Pull- and Push-Based Data Transfer

A major difference between batch processing (analysis of data at rest) and stream processing (real-time analysis) is the way data transfers are initiated. MapReduce [13] systems and relational databases process previously stored data when they execute a query. Thus, they can pull data from disk as needed, for example, using the iterator model. In contrast, stream processing systems have no control over incoming streams, which can push data into the system at an arbitrary rate.

We combine push- and pull-based data transfer: on the one hand, we pull data from sensors[1] based on the data demand of queries. On the other hand, we asynchronously push data through the stream processing pipeline, which enables low latency processing.

UDSFs and our *read scheduling algorithm* are applicable wherever data is pulled from a source. This, for example, also holds for service APIs such as *Twitter Streaming* or *Google Cloud Prediction*. Avoiding oversampling on these APIs directly results in financial savings because charges apply per API call [20].

3.2 Adaptive Sampling

Adaptive sampling techniques such as AdaM [41], FAST [17], and L-SIP [19] reduce oversampling compared to periodic sampling. They reduce sampling rates on the fly whenever values evolve predictably or remain constant. At the same time, they increase sampling rates as required, to not exceed failure tolerances.

Different use cases require different adaptive sampling techniques: for example, AdaM [41] is robust against abrupt value fluctuations and provides good approximations of time series. FAST [17], on the other hand, incorporates concepts of differential privacy for real-time aggregate monitoring. Our read scheduler allows for multiplexing different adaptive sampling algorithms in parallel on shared sensors to enable reduced average sampling rates.

We implement AdaM and FAST as examples for adaptive sampling techniques. Both combine adaptive sampling with adaptive filtering. However, the algorithms differ fundamentally from each other: AdaM uses *Probabilistic Exponential Moving Averages* [9] for value estimations. In contrast, FAST adopts a *Proportianal Integral Derivate* controller [28].

[1] We refer to physical sensors, such as photo cells or accelerometers, as *sensors*, and call the devices which host sensors *sensor nodes*.

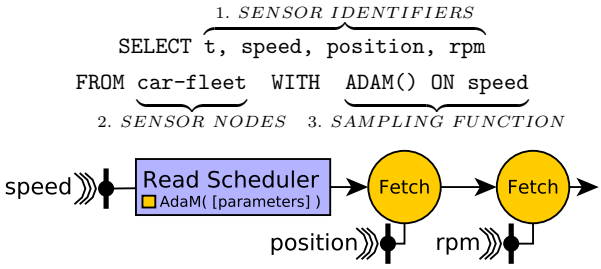


Figure 3: An example query with user-defined sampling and its corresponding processing pipeline.

3.3 The User's Perspective

It is important to highlight that the complexity of multi-query read scheduling is transparent to users. Users can still define streaming queries in declarative languages such as CQL [3] or SPL [21].

From the perspective of a user, a query is executed over a data stream consisting of tuples $(t, s_1, s_2, ..., s_n)$ where t is the timestamp of a tuple and s_1 to s_n are the values from all available sensors at time t. This is a common data model in stream processing systems.

We show an example query with its corresponding processing pipeline in Figure 3. The query acquires data to compute a driver profile in accordance to our introductory example (Figure 1). We omit a more complex profiling algorithm for the sake of simplicity.

The query consists of three parts: *(i)* Sensors are referenced by identifiers similar to column names in SQL. *(ii)* Instead of tables, we refer to sensor nodes as data sources in the `FROM` clause. *(iii)* We add a `WITH` clause to specify a UDSF and the sensor it is applied to. The user specifies the data demand of the query by implementing a UDSF or choosing a pre-defined one. This empowers domain experts to express their data demand flexibly and also enables adaptive sampling techniques. We will explain UDSFs in detail in Section 5.

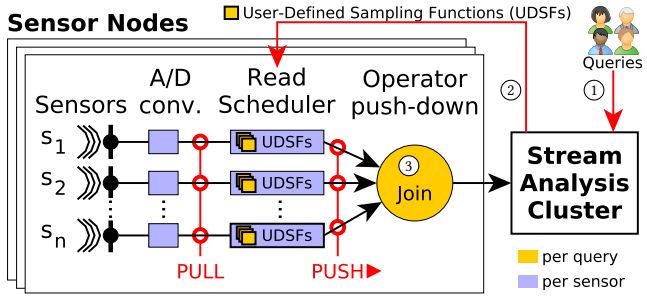
The processing pipeline of the query starts with the read scheduler, which uses AdaM to sample the speed sensor. It then fetches the position and the revolutions per minute (rpm) in an ad-hoc fashion in order to construct the output tuples. This is regularly beneficial because we reduce sampling rates in comparison to periodic sampling with a constant rate.

4 SYSTEM ARCHITECTURE

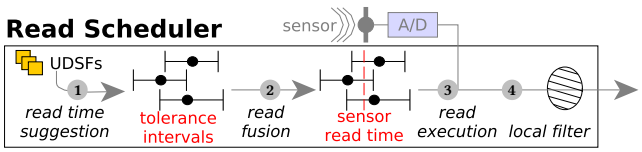
In this section, we present how *on-demand streaming from sensor nodes* eliminates unneccessary sensor reads and thus, data transmissions. In Figure 4a, we illustrate how *on-demand streaming* integrates with streaming systems.

First, users submit their queries and their data demand (expressed by *UDSFs*) to a stream analysis cluster ①. We then propagate the UDSFs to the sensor nodes ②.

For each sensor, we perform *read scheduling* in four phases (Figure 4b): *read time suggestion*, *read fusion*, *read execution*, and *local filtering*. First, during *read time suggestion* **1**, each UDSF (provided with each query) proposes a read time with a tolerance interval. Second, during *read fusion* **2**, we fuse proposed read times to a single sensor read, if the tolerance intervals overlap. Third, during *read execution* **3**, we perform the actual read on the sensor. Finally, during *local filtering* **4**, we determine if we need to transmit the



(a) Overall on-demand streaming architecture.



(b) Read scheduler internals.

Figure 4: On-demand streaming architecture.

obtained sensor value. We can, e.g., avoid transmitting values which are similar to previous ones or follow an expected trend.

Read time suggestion allows for *adaptive sampling* to avoid oversampling. This is especially important whenever charges apply per read (e.g., service API calls). *Read fusion* avoids redundant data transmissions and enables sensor read sharing among queries. It thereby reduces network charges. *Local filtering* further reduces data transmissions, which reduces the inbound traffic at the analysis cluster and prevents scalability challenges.

Complementary Techniques. Our scheduler works complementary to the succeeding push-based processing pipeline (Figure 4a ③), which can consist of arbitrary stream transformations such as aggregations, filters, or stream joins [2, 11, 27]. It thereby goes hand-in-hand with techniques such as query fusion on sensor nodes [30, 44, 45], operator push-down, and *acquisitional query processing (ACQP)* [26]. The combination with ACQP is of special interest: we first apply read scheduling on a subset of sensors to avoid oversampling. We then further reduce the data with filters and aggregations. Finally, we fetch values from additional sensors for the remaining tuples only. We will discuss all mentioned techniques in more detail when presenting related work in Section 8.

Alternative Architectures. In this paper, we study a setting were we execute read scheduling on sensor nodes. We tested our algorithms using Raspberry Pis and Android smart phones as sensor nodes and did not experience any performance problems. However, our read scheduler also works as a middleware layer which aggregates UDSFs (i.e. queries) at a more powerful machine close by the sensor nodes (i.e. a base station server or a router). Our read scheduler pulls values from sensors (i.e. it samples the sensor) on the fly based on the data demand expressed in UDSFs. This enables adaptive sampling but required a low latency connection between the read scheduler and the sensor we sample.

5 USER-DEFINED SAMPLING

Different applications have contradicting sampling requirements. They vary in sampling rates, transfer different fractions of sensor values, and have different requirements for read time precision and data freshness (maximum age of values arriving at the cluster).

User-defined sampling functions (UDSFs) allow for the precise definition of each query's data demand and facilitate *adaptive sampling* techniques. This makes them the basis for avoiding *oversampling*. They further model read time tolerances and preferences, which enables *read fusion* to solve the *redundant transmission problem*.

In the following section, we first discuss how we *enable read fusion* and *optimize sensor read times*. This leads to our model for the read times proposed by UDSFs (in short *read requests*). We then show how we can cover example applications with *UDSFs*. Finally, we introduce local filter functions to further reduce data transmissions.

5.1 Enabling Read and Traffic Sharing

Sampling techniques define exact times where values shall be read from sensors. The probability that we can fuse two requested reads (share sensor reads among queries) decreases with the read time precision and vice versa. In order to enable frequent read fusion, applications have to specify their precision requirement for read times. We thus represent requested sensor reads (in short: *read requests*) as tolerance intervals instead of exact times. We share sensor reads as well as the corresponding traffic among queries whenever tolerance intervals overlap.

For many use cases, a certain deviation from the desired read times (*read time slack*) is possible without harming the result quality. For example, consider a query which requires the current temperature every hour. This query does not require a nanosecond read time precision but can offer a tolerance, e.g., one minute. We found that sophisticated adaptive sampling techniques such as AdaM [41] and FAST [17] are robust against a certain slack in read times as we show in our experiments in Section 7.2.3. We further argue that reading before the desired time is regularly harmless as it improves the data quality upon a given minimum sampling rate.

5.2 Global Read Time Optimization

Our scheduling algorithm not only minimizes the number of sensor reads, it also optimizes the exact sensor read times. We provide semantics to model read time preferences by introducing penalty functions ($p(t)$). Each read request can thereby define its individual penalty function.

For example, consider our introductory use case in Figure 1: Query 2 (Route Profile) requires a sample at least every 20 driven meters. Reading earlier is harmless and we can thus define our penalty function as $p(t) = 0$ (i.e., we do not apply any penalty for read time deviations). At the same time, we execute Query 1 (Driver Profile), which uses AdaM. In this case, read time slack might affect the result quality and thus we set $p(t) = t^2$. In general, we can set any penalty function which describes our read time preferences. In case of Query 1, we choose the quadratic function t^2 to avoid large deviations by penalizing them much more than smaller deviations.

In our example, the read time optimizer freely decides for a read time within the tolerance intervals of Query 2, because no penalty

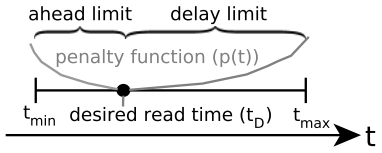


Figure 5: Read request with desired read time, tolerance interval, and a convex penalty function.

applies for deviating from the desired read time. At the same time, the optimizer minimizes the deviation from the desired read time for Query 1 to avoid the penalty of t^2.

Our optimizer minimizes the sum of the penalty functions for overlapping tolerance intervals. Thereby, it determines the next sensor read time. In order to enable the minimization at low computational costs, we require all penalty functions to be convex and to have their minimum at the desired read time ($t_D = 0$). We further shift penalty functions along the y-axis such that $p(t_D) = 0$. We present the optimization process in detail in Section 6.

5.3 Modelling Read Requests

As a result of the considerations from the previous sections, we model *read requests* as illustrated in Figure 5:

- Each requested read is described by a tolerance interval $[t_{min}, t_{max}]$, which covers the desired read time t_D.
- The distance between t_{min} and t_D is the tolerance for reading ahead of t_D. Respectively, the distance from t_D to t_{max} is the tolerance to delay the read.
- Within each interval, read time preferences are modelled with a penalty function $p(t)$.

Our scheduling algorithm first minimizes the total number of executed sensor reads based on interval overlaps. It then optimizes the exact read times based on the given penalty functions. UDSFs can adjust read time tolerances and penalty functions individually for each read request.

5.4 User-Defined Sampling Functions

Syntax. Formula 1 shows the structure of a *user-defined sampling function (UDSF)*. Upon a sensor read, the function receives the current timestamp t and the current sensor value v. In exchange, it returns a tuple $\langle t_{min}, t_D, t_{max}, p(t)\rangle$. The output tuple corresponds to our model for read requests and consists of the next desired read time t_D, the tolerance interval $[t_{min}, t_{max}]$, and the penalty function $p(t)$.

$$s : \langle t, v\rangle \rightarrow \langle t_{min}, t_D, t_{max}, p(t)\rangle$$

Formula 1: User-defined sampling function.

At any time, we only require the next read request from a sampling function. This allows for adapting sampling rates, read time tolerances, and penalty functions flexibly after each sensor read. We allow sampling functions to keep a state because many sampling techniques need to remember previous sensor values or variables.

Examples. The presented sampling function is easy to implement and facilitates various use-cases. Let us first consider our introductory example (Figure 1). Example 1 shows the sampling function serving Query 1 (Driver Profile). It also shows how the AdaM algorithm, as a representative for adaptive sampling functions, can be integrated in a UDSF. The call to the AdaM algorithm in Line 2 can be replaced with any other adaptive sampling algorithm. The shown implementation constantly applies a read time tolerance of $\pm 0.2s$ and a linear penalty function $p(t) = |t|$.

```
1: upon sensor read ⟨time, value⟩ do
2:     t_D ← AdaM(time, value) // get next read time
3:     t_min ← max(time, t_D − 0.2s) // get ahead limit
4:     t_max ← t_D + 0.2s // get delay limit
5:     p(t) ← abs(t − t_D) // set penalty function
6:     return ⟨t_min, t_D, t_max, p(t)⟩
7: end
```

Example 1: AdaM with 0.2s read time tolerance.

One major advantage of user-defined sampling is the ability to adapt sampling rates driven by the values gathered before. Query 2 (Route Profile) from Figure 1 is an example for a case where we need an application specific data-driven sampling function: we require a value for at least every 20 meters driven. With periodic sampling, we would need to always assume the maximum speed of the car and set the time between two sensor reads to be $\frac{20m}{max(v)}$. However, cars seldom drive with their maximum speed and periodic sampling would cause *oversampling* during all the remaining time.

$$s_{20m} : \langle t, v\rangle \rightarrow \langle t+1, t+\tfrac{20m}{v}, t+\tfrac{20m}{v}, 0\rangle$$

Example 2: Sample at least every 20 driven meter.

In contrast to periodic sampling, our user-defined function in Example 2 can calculate the next read time based on the current speed upon each sensor read. We further configured t_{min} as the current timestamp plus 1, meaning that we subscribe to any sensor read, which will be executed before we passed 20m. Note that the added tolerance can only decrease the total number of executed sensor reads. The scheduler always prefers t_D over any other time in $[t_{min}, t_{max}]$. The scheduler will only utilize the tolerance in case a sensor read must be executed anyways to serve another query.

$$s_{0.3s} : \langle t, v\rangle \rightarrow \langle t+1, t+0.3s, t+0.3s, 0\rangle$$

Example 3: Read a value at least every 0.3s.

With Example 3, we address Query 3 from Figure 1. This query samples periodically with the same ahead limit as the previous example. This example emphasizes the compatibility of our approach with common periodic sampling. Our read scheduler seamlessly combines periodic sampling functions with more advanced sampling functions such as the ones in Example 1 and 2.

5.5 Local Filter Functions

As an additional optimization, we couple our UDSFs with local filter functions (Formula 2). Local filtering allows for further reducing data transmissions.

$$f : \langle t, v\rangle \rightarrow \{true, false\}$$

Formula 2: Local filter function.

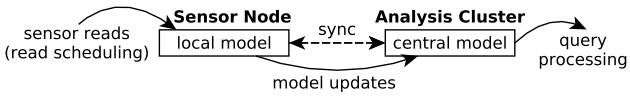


Figure 6: Model-driven data acquisition.

```
1: upon sensor read ⟨time (t), value (v)⟩ do
2:     mv ← model.estimateValue(t)
3:     if abs(mv − v) > tolerance then
4:         model.update(t, v) // local model update
5:         return true // transfer value
6:     else
7:         return false // no transfer required
8:     end if
9: end
```

Example 4: Local filter for model-driven data acquisition.

For example, we do not transfer sensor values if they remain constant or follow an expected trend.

Similar to the sampling function, the filter function is called upon a sensor read with the current time and sensor value as parameters. It returns a boolean value, which indicates if the current measurement shall be transferred upstream. UDSFs and filter functions can communicate through a shared state.

Model-driven data acquisition (Figure 6) is an example for local filtering [15, 33]. This technique estimates sensor values using a model, which is based on previously gathered values (e.g., regression techniques or pattern learning). As shown in Example 4, the filter function compares sensor values with the model-based estimation. No data transmission is required if the difference lies within a failure tolerance i.e. the central model is sufficient.

We refer the reader to the original works for detailed descriptions and throughout evaluations of the diverse adaptive filtering techniques available [10, 15, 22, 33, 41, 42].

6 MULTI-QUERY READ SCHEDULING

Each query can define its own *UDSFs*. Accordingly, several different *UDSFs* can be present at a single sensor that is shared among queries. A naive approach would execute each UDSF separately and miss the opportunity to share sensor reads and data transmissions among them. We contribute an algorithm that exploits read time tolerances to share sensor values among multiple queries. Our multi-query read scheduling algorithm minimizes the number of sensor reads with respect to query needs. It further optimizes the exact read times with respect to the given penalty functions, while still performing the minimum number of sensor reads only.

6.1 Minimizing Sensor Reads

Our primary goal is to minimize the number of performed sensor reads. To that end, each *UDSF* suggests a read time in the form of a *read request* (Section 5.3). We then apply *read fusion* to combine *read request* with overlapping tolerance intervals. This maximizes read and traffic sharing among queries and minimizes sensor reads. Our algorithm is agnostic to the underlying algorithms of UDSFs. It solely operates based on the provided *read requests*.

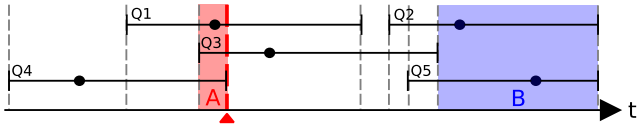


Figure 7: The latest possible time for the next read is the first interval end. Reading at this time minimizes the total number of sensor reads because the sharing potential increases up to this point and reading later is impossible.

Guaranteed minimum of sensor reads. We present a read scheduling algorithm, which guarantees to perform the minimum number of sensor reads only.

Initially, during *read time suggestion*, all present UDSFs provide their next *read request*. We then minimize the number of sensor reads using *read fusion*. In Figure 7, we show an example for the *read fusion* phase, where five UDSFs provide their *read requests*. Given the read requests, we can determine the latest possible time for the next sensor read: it is the first end of any tolerance interval (red dashed line). Reading later would violate the read time tolerance of Q4 and is thus impossible. Reading earlier can only decrease the amount of fused read requests because only interval starts can lie before the first interval end. This leads to the important observation that *reading at the time of the first interval end minimizes the number of sensor reads.*

Once we perform the sensor read at the end of the Q4 interval, we can share the obtained value among three queries: Q1, Q3, and Q4. Our scheduling algorithm then acquires the next *read requests* from the *UDSFs* of Q1, Q3, and Q4. It keeps the intervals from Q2 and Q5 because they start in the future. Given all *read requests*, we repeat the described process to schedule the next read.

6.2 Optimizing Read Times

Our secondary goal is to optimize the deviation from desired read times, while still executing the minimum number of sensor reads only. Hence, we extend the *read fusion* phase of our algorithm with *read time optimization*.

Preliminary Considerations. We divide the time axis in non-overlapping time intervals, which we call *fragments*. Each start and each end of a tolerance interval is thereby considered as fragment separator. For example, consider Figure 7, where fragments are separated with dashed lines. The used fragmentation technique is known as stream slicing [8, 23, 24] and is widely used in streaming window aggregation. We use Cutty [8] to derive fragments within a single-pass over the tolerance intervals.

The number of overlapping intervals - and thereby the read sharing potential - remains constant within fragments. This is the case because each start or end of a tolerance interval, that changes the number of overlapping intervals, also marks the start of a new fragment. We thus perform the minimum number of sensor reads as long as we perform sensor reads in the last fragment before the first end of any tolerance interval. For example, consider the red shaded fragment in Figure 7.

The Optimal Fragment. As a result of our preliminary considerations, we aim to optimize the read time within the latest

Algorithm 1 Get the optimal fragment for the next read.

Parameter:
$rInt[\,]$: Array of read requests $\langle t_{min}, t_D, t_{max}, p(t)\rangle$.
Output:
The optimal fragment for the next sensor read.

```
1: function GetOptimalFragment(rInt)
2:     t_end ← min(t_max) from rInt
3:     t_start ← max(t_min) from rInt where t_min ≤ t_end
4:     return [t_start, t_end]
5: end function
```

Algorithm 2 Read time optimization.

Parameter:
$rInt[\,]$: Array of read requests $\langle t_{min}, t_D, t_{max}, p(t)\rangle$.
Output:
The optimized timestamp for the next sensor read.

```
1: function OptimizeReadTime(rInt)
2:     [t_start, t_end] ← GetOptimalFragment(rInt)
3:     rInt ← AssignIntervals(rInt,t_start,t_end)
4:     return MinimizePenalty(rInt,t_start,t_end)
5: end function
```

fragment before the first end of any tolerance interval. This guarantees executing the minimum amount of sensor reads, but reduces the deviations from the desired read times.

Algorithm 1 formalizes how we determine the optimal fragment in which we can optimize the exact read time. The optimal read time within the optimal fragment is the time which implies the smallest penalty.

Read Time Optimization. We summarize the overall process of the read time optimization in Algorithm 2. We first call Algorithm 1 to get the optimal fragment. We then decide in Line 3 for which *read requests* we will use the next sensor value. This, for example, removes tolerance intervals which start after the selected optimal fragment (e.g., Q4 and Q5 in Figure 7). We finally minimize the penalty within the optimal fragment and return the read time.

The penalty at any time is given by the sum of the penalty functions of all tolerance intervals being present at this time. Since each penalty function is convex, their sum $p_\Sigma(t)$ is also a convex function [35], which has a single minimum only. We can find this minimum (giving the optimal read time) with $O(log(\frac{l}{\Delta}))$ complexity, where l is the length of the optimal fragment $[t_{start}, t_{end}]$ and Δ is the length of the confidence interval. We therefore initialize the confidence interval with $[t_{start}, t_{end}]$. We then calculate the derivative $p'_\Sigma(x)$ with x being the center of $[t_{start}, t_{end}]$. If $p'_\Sigma(x) = 0$, x is the minimum. Otherwise, the sign of $p'_\Sigma(x)$ denotes if x lies left or right of the minimum. If x lies left, we assign $t_{start} \leftarrow x$, otherwise $t_{end} \leftarrow x$. While repeating the process, we half the confidence interval with each iteration until $t_{end} - t_{start} < \Delta$.

Assigning Read Requests to Fragments. In order to optimize read times, we need to assign *read requests* to the optimal fragment in which we perform the next sensor read (Line 3 in Algorithm 2). The read time optimization within the optimal fragment is then based on the penalty functions of the assigned read requests only.

So far, we just considered the first upcoming read, but not the succeeding ones. In the remainder of the paper, we call the optimal

Algorithm 3 Assign read requests to selected fragments.

Parameters:
$rInt[\,]$: Array of read requests $\langle t_{min}, t_D, t_{max}, p(t)\rangle$.
$[t_{start}, t_{end}]$: The optimal interval for the next read.
Output:
$rInt[\,]$: Read requests assigned to the next read.

```
1: function AssignIntervals(rInt, t_start, t_end)
2:     rInt' ← all r ∈ rInt where r.t_min > t_end
3:     [t'_start, t'_end] ← GetOptimalFragment(rInt')
4:     for each r ∈ rInt
5:         if [t_start, t_end] ⊈ r then remove r from rInt
6:         else if [t'_start, t'_end] ⊈ r then keep r in rInt
7:         else if t_end > r.t_D then keep r in rInt
8:         else if t'_start < r.t_D then remove r from rInt
9:         else if r.p(t'_end) < r.p(t_end) then
10:            remove r from rInt (Figure 9a)
11:        else if r.p(t_start) < r.p(t'_start) then
12:            keep r in rInt (Figure 9b)
13:        else remove r from rInt (Figure 9c)
14:        end if
15:    end for each
16:    return rInt
17: end function
```

Definition: Let r be a *read requests* $\langle t_{min}, t_D, t_{max}, p(t)\rangle$ and i be an interval $[t_{start}, t_{end}]$. We then say that $r \subseteq i$ if $[r.t_{min}, r.t_{max}] \subseteq i$.

fragment for the next sensor read **A**, and the latest possible fragment for the second sensor read **B**.

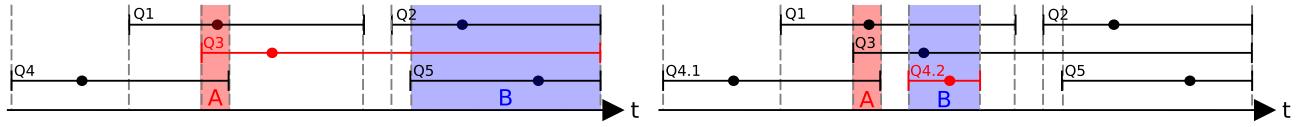
Assigning read requests to fragments is not always straight forward. We show the trivial case in Figure 7. Each tolerance interval covers only one selected optimal fragment. Accordingly, we assign read requests either to the first read (Fragment A) or the second read (Fragment B). This example changes in Figure 8a. The Q3 tolerance interval now covers both, the first (A) and the second (B) selected fragment. In case we assign Q3 to Fragment A, it will not affect the read time optimization for Fragment B and vice versa.

We present the assignment process, including the non-trivial cases, in Algorithm 3. The algorithm first determines the latest possible fragment for the second read, which is marked blue in Figures 7 and 8a. Therefore, our algorithm defines $rInt'$ as an array of all read requests, which cannot be assigned to A (Line 2). It then calls Algorithm 1 as subroutine with $rInt'$ as parameter to determine fragment B (Line 3).

In the special case, that all read requests can be assigned to A, $rInt'$ is empty in Algorithm 3. B is thus undefined and we assign all read requests to A. In the regular case, where we can compute A and B, we differentiate between seven cases to decide if we assign a tolerance interval to Fragment A. Intervals which are not assigned to A will get assigned to other fragments upon the optimization of subsequent read times.

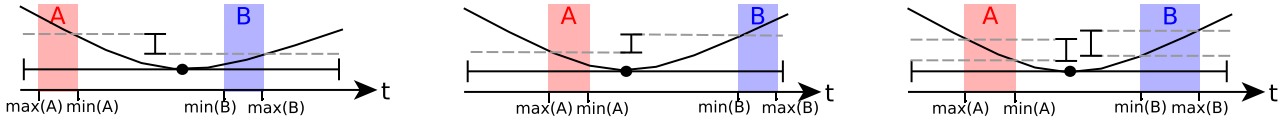
Case 1: No overlap with A. We cannot assign tolerance intervals to A, which do not overlap with A (Line 5). This would violate the read time tolerance.

Case 2: No overlap with B. We assign tolerance intervals to A, which do not overlap with B (Line 6). This ensures that such tolerance intervals cannot cause additional sensor reads before B. This retains the guarantee to execute the minimum number of sensor reads only.



(a) Tolerance intervals possibly cover several read operations. (b) New read requests may cause earlier read times.

Figure 8: Challenges in the assignment of read read requests to selected fragments in which we perform sensor reads.



(a) max(B) < min(A) ⇒ **postpone.** (b) max(A) < min(B) ⇒ **assign to A.** (c) **otherwise ⇒ postpone.**

Figure 9: Deciding for a fragment in case a tolerance interval overlaps with several sensor read times.

Case 3: t_D before end of A. We assign tolerance intervals to A, which have their desired read time before the end of A (Line 7). This is sure to be optimal because the penalty can only increase towards B in this case.

Case 4: t_D after start of B. We do not assign tolerance intervals to A, which have their desired read time after the start of B (Line 8), because the penalty decreases towards B.

The remaining cases are shown in Figure 9. Both, A and B, overlap with the tolerance interval. A must be before the desired read time, and B after the desired read time.

Fragment B is the latest possible time for the second read. However, it is important to highlight that B is subject to change: after the first read is performed, all *UDSFs*, whose read requests were assigned to A, provide their next read requests. The corresponding new tolerance intervals possibly end before B, which moves B closer to A. For example, consider Figure 8b. The tolerance interval Q4.2 appears after A and causes B to shift towards A.

Due to our limited knowledge about the second read time - we only know that it wont be later than B - we cannot guarantee that our assignment is optimal. Nonetheless, we propose a best effort approach based on the minimum and maximum values of the penalty in A and B:

Case 5: $max(B) < min(A)$. We do not assign tolerance intervals to A for which the penalty in B is always smaller than the penalty in A (Line 9/Figure 9a). In this case, it is guaranteed that there will be another read after A with reduced penalty.

Case 6: $max(A) < min(B)$. We assign tolerance intervals to A in case the penalty is always smaller in A than in B (Line 11/Figure 9b). This decision is not guaranteed to be optimal because B could possibly shift closer to A. However, A is regularly quite close to the desired read time when this condition holds true.

Case 7: otherwise. We do not assign tolerance intervals to A in case there is an overlap in the penalties of A and B (Line 13/Figure 9c). The penalty in B can still reduce when B moves towards A. In case it does not, we can get the same penalty in B as we could in A.

We now have all pieces at hand, which we require for our overall scheduling algorithm: *(i)* we can select optimal fragments in which we perform sensor reads, *(ii)* we can smartly assign read requests to the optimal fragments, and *(iii)* we can minimize the penalty for the next sensor read time.

Algorithm 4 The overall scheduling algorithm.

State:
$udsf[\,]$: Array of *user-defined sampling functions.*
$rInt[\,]$: Array with next read requests from all *UDSFs* in the form $\langle t_{min}, t_D, t_{max}, p(t)\rangle$.
Output:
The timestamp of the next sensor read.

```
1:  upon sensor read ⟨t, v⟩ do
2:      [t_start, t_end] ← GetOptimalFragment(rInt)
3:      rInt_now ← AssignIntervals(rInt,t_start,t_end)
4:      for i from 0 to udsf.size − 1 do
5:          if rInt[i] ∈ rInt_now then
6:              // Apply local filter of udsf[i]
7:              if udsf[i].f(t, v) then
8:                  subscribe udsf[i] to current read ⟨t, v⟩
9:              end if
10:             // next read request for udsf[i]
11:             rInt[i] ← udsf[i].s(t, v)
12:         end if
13:     end for
14:     transmit current read ⟨t, v⟩ to subscribers
15:     return OptimizeReadTime(rInt)
16: end
```

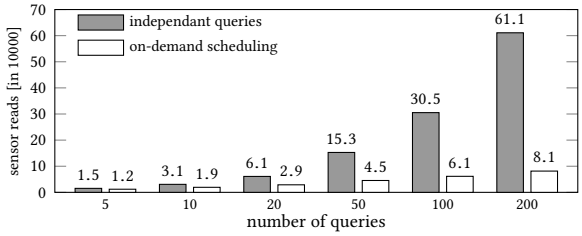
6.3 The Overall Scheduling Algorithm

The overall read scheduling algorithm (Algorithm 4) operates based on the *UDSFs* present at a sensor. It is called upon each sensor read and returns the time of the next sensor read. It further applies the local filter functions and initiates the transfer of the sensor values.

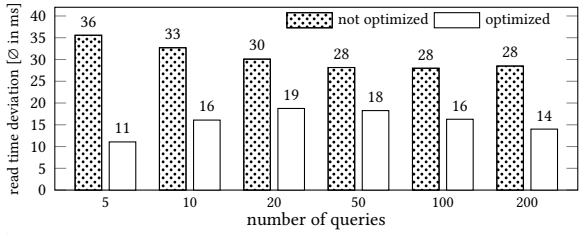
At start-up time, we perform one initial sensor read and pass it as parameter to all *UDSFs* to obtain their first read requests. This initializes the *rInt* array with read requests from all *UDSFs*. When we add a new*UDSF*, the scheduler requests the next read request from the new *UDSF* with the previous sensor value as parameter. We omit this initialization process in Algorithm 4.

Each subsequent sensor read is processed in four steps:

1. In Line 2 and 3, we assign read requests to the current sensor read using Algorithms 1 and 3.
2. For each read request, which is assigned to the current sensor read, we apply the local filter of the corresponding *UDSF* (Line 5). In case the value passes the filter, we subscribe the *UDSF* to the



(a) Number of sensor reads and data transmissions.



(b) Impact of read time optimization on read time deviations.

Figure 10: Increasing the number of queries. (random UDSFs; ∅sampling rate 1Hz/UDSF; ∅tolerance ±0.04s)

upcoming data transmission (Line 8). In any case, we acquire the next read request and store it in the *rInt* array (Line 11).

3. We initiate the data transmission of the current sensor value to all subscribers (Line 14). This happens through an asynchronous function call to not delay the computation of the next read time.
4. Finally, we call OptimizeReadTime(*rInt*) (Algorithm 2) and return the time for the next sensor read.

Note that the calls to GetOptimalFragment(*rInt*) and AssignIntervals($rInt, t_{start}, t_{end}$) within Algorithm 2 are redundant to the calls in the first step (Line 2 and 3) of Algorithm 4. An efficient implementation would keep the assignment as state to prevent doubled computation. We omit this optimization to simplify the exposition.

7 EXPERIMENTAL EVALUATION

In this section, we evaluate on-demand streaming from sensor nodes on real-world sensor data. We first present our experimental setup, then show our results, and close with a discussion.

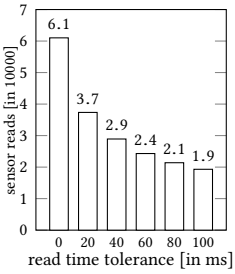
7.1 Experimental Setup

Data. We replay recorded sensor data from two datasets: First, the *Formula 1* telemetry data which we introduced in our introductory use-case in Section 2. Second, sensor data from a *football* match which was provided with the DEBS'13 Grand Challenge [31]. We monitor the speed of the ball, which is tracked with a 2000Hz sampling rate and µm/s precision.

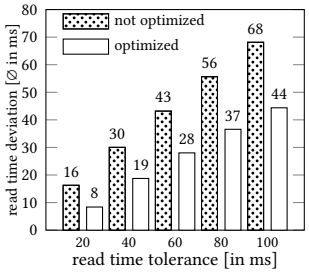
Workloads. Our experiments use three query sets:

Introductory use-case: We presented an initial evaluation of our introductory use case in Section 2. We use AdaM as adaptive sampling technique in combination with the UDSF from Example 2 and periodic sampling.

Random UDSFs: We use queries with random UDSFs to study the scalability of our solution to large numbers of concurrent queries and users. In our experiments, one UDSF corresponds to one query.



(a) Sensor reads/transfers.



(b) Read time optimization.

Figure 11: Increasing the read time tolerance. (20 queries, i.e., 20 random UDSFs, ∅sampling rate 1Hz/UDSF)

Thus, the number of UDSFs and the number of queries are the same. In general, queries can define multiple UDSFs to request data from several sensors. Our scheduling algorithm solely operates based on the UDSFs and is agnostic to all other query properties.

Our random UDSFs submit read requests in a *Poisson process*. Poisson processes [12] are widely used in statistics to model independent random events such as starts of phone calls [6]. Read requests are similar to phone calls: they may occur at any time, have peak times, and periods of low utilization. The lengths of tolerance intervals are exponentially distributed. Thus, small read time tolerances are most frequent. The probability for larger tolerances decreases exponentially. Whenever we use random UDSFs in our evaluation, we apply the distributions described above.

AdaM and FAST: We execute AdaM and FAST individually to examine their robustness against read time slack. This verifies that read time tolerances do not harm the result quality of adaptive sampling techniques.

7.2 Detailed Experiments

We analyzed the number of sensor reads and transferred tuples for our introductory use-case in Section 2. In the following section, we show that our solution also scales to larger query sets. Therefore, we compare our on-demand data streaming approach with an independent execution of multiple queries. Then, we evaluate the achievements of our read time optimizer. Finally, we investigate the impact of read time slack on different sampling strategies.

7.2.1 Shared Sensor Reads and Traffic.

Scaling the Query Set. *On-demand* scheduling scales to larger query sets. We increase the number of queries up to 200 in Figure 10a. Increasing the number of queries is equivalent to increasing the sampling frequency of queries: our read scheduler solely operates based on submitted read requests. Thus, the number of read request makes the difference rather than the number of queries.

Periodic sampling is virtually impossible in this experiment: UDSFs read in a Poisson process, which simulates heavy peaks in sampling rates. Periodic sampling would fall back to the maximum sampling rate, which is in the order of 10^9Hz. Hence, we compare an independent query execution with our on-demand streaming approach.

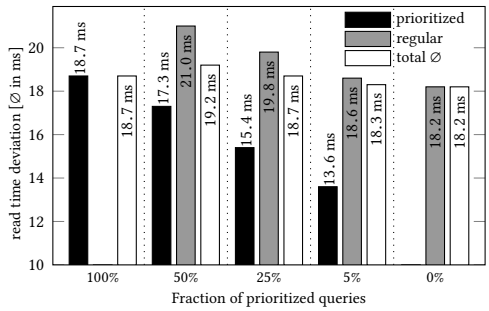


Figure 12: Query prioritization with penalty functions. (20 queries; ∅sampling rate 1Hz/-query; ∅tolerance ±0.04s)

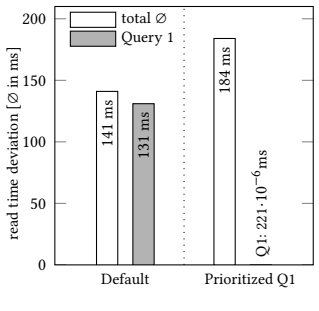
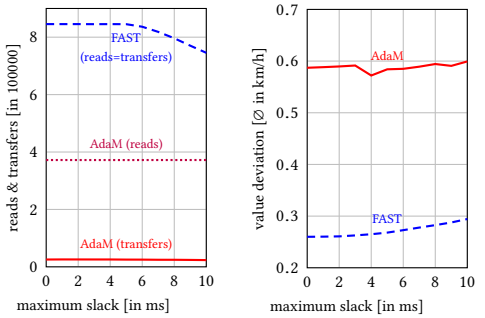


Figure 13: Read time optimization on behalf of a single query. Introductory use-case.



(a) Reads & Transfers **(b) Value Deviation**

Figure 14: AdaM and FAST on football data with varying read time slack.

For the *independent execution of queries*, the number of sensor reads and data transmissions increases linearly with the number of queries. This is because each read request causes a sensor read and a transmission.

On-demand scheduling can fuse read requests whenever their tolerance intervals overlap. The probability for such overlaps increases with the number of read requests. Thus, read and traffic sharing becomes more frequent with larger query sets. We increase the number of queries by factor 40. However, the number of reads increases by less than factor 7, saving 87% in reads and transfers.

Increasing Tolerances. Another way to increase the probability for read fusion is to increase read time tolerances. We analyze this effect in Figure 11a. We therefore fix the number of queries to 20. The number of sensor reads decays exponentially when the tolerance increases. This observation is in accordance with the theoretical coincidence probability of random events with exponentially distributed lengths described by Erlang et al. [6].

7.2.2 *Read Time Optimization.*

We now evaluate the deviation from desired read times in our experiments. Our read time optimizer never increases the amount of sensor reads or transfers. However, it reduces the mean deviation from desired read times by up to 69% in our experiment with larger query sets (Figure 10b).

We observe two contradicting effects in Figure 10b: On the one hand, more read requests increase read time deviations. The probability of read sharing increases and we utilize tolerances to fuse reads. This effect dominates up to 20 concurrent queries. On the other hand, more read requests decrease read time deviations: It becomes more probable that multiple sensor reads take place within the tolerance interval of a read request. In such cases, the optimizer selects the sensor read which implies the smallest read time deviation. This effect dominates for 50 or more concurrent queries.

In Figure 11b, we study how an increasing read time tolerance affects the optimization. The read time deviation increases with the read time tolerance, because we use additional tolerances primarily to reduce sensor reads and data transmissions. Thus, the selected fragments, in which we perform the optimization, deviate more from the desired read times of read request.

Query Prioritization. Each UDSF can define its individual penalty function to model read time preferences within tolerance intervals. We use this feature to prioritize selected UDSFs when optimizing read times. For example, prioritized UDSFs may penalize read time deviations with $p(t) = t^2$, while non-prioritized UDSFs set $p(t) = |t|$. We analyze the impact of such a prioritization in Figure 12. Prioritization reduces read time deviations considerably for the prioritized UDSFs. This effect declines when the fraction of prioritized UDSFs increases. When many UDSFs are prioritized, sensor reads are often shared among them which repeals the prioritization. The read time deviation for non-prioritized UDSFs increases with the fraction of prioritized UDSFs. The same holds for the overall mean deviation. Hence, we recommend to prioritize small subsets of UDSFs only.

Prioritizing all queries (100%) leads to a mean read time deviation of 18.7ms. Prioritizing no query (0%) reduces the mean read time deviation to 18.2ms. This is because $p(t) = |t|$ (not prioritized) grows linear when the read time deviation increases. This minimizes the overall sum of read time deviations and, thereby, the mean read time deviation. In contrast, $p(t) = t^2$ (prioritized) grows quadratically and focuses on avoiding high deviations rather than minimizing the mean deviation.

We consider the example from Figure 12 as being a gentle prioritization. We can of course apply more strict differentiations between UDSFs by increasing the differences between penalty functions. For example, by multiplying the functions or by increasing the power. Our introductory use-case is an extreme yet realistic example for UDSF prioritization: the AdaM UDSF tolerates read time slack, but with rather high penalty of $p(t) = t^2$. Other queries forbid any read time delay, but are fine with reading earlier. The penalty for reading ahead of the desired read time is thus zero. Accordingly, we optimize read times solely on behalf of AdaM, resulting in a mean deviation in the order of nanoseconds (Figure 13).

7.2.3 *The Effect of Read Time Slack.*

Our experiments show that read time tolerances lead to fewer sensor reads and transferred tuples. Hence, we advocate read time tolerances for adaptive sampling techniques. We now analyze how read time deviations affect AdaM and FAST, our representatives of adaptive sampling techniques. Therefore, we affect sensor reads by

uniformly distributed random slacks. In this section, we monitor the speed of the ball during a football match.

We shows the number of sensor reads and transfers for different read time slacks in Figure 14a. Both, AdaM and FAST, are robust against slack: the number of sensor reads and transfers remains almost constant for slacks up to ±5ms. Larger slacks reduce sensor reads for FAST, because read time delays can be larger than the average read frequency. The adaptive filter of AdaM massively reduces data transfers as it avoids sending consecutive similar measures when the football is on the ground or airborne.

The mean deviation between the obtained speed graph and the underlying DEBS'13 raw data increases slightly with the slack (Figure 14b). However, we consider both techniques as robust because they retain a mean deviation of less than 0.6 km/h on the volatile speed of a football.

7.3 Discussion

On-demand streaming from sensor nodes reduces the number of sensor reads and the amount of transferred data by 57% in our introductory use-case and by up to 87% with larger query sets (i.e. more read requests per time). In comparison, periodic sampling leads to extremely high sampling rates, because it falls back to the maximum sampling rate required at any time. Adaptive sampling reduces sensor reads for a single query, but falls short in combining multiple different data demands. On demand sampling unites adaptive sampling with the multiplexing of different data demands, which explains the savings.

In our experiments, the read time optimizer reduces the mean deviation from desired read times by up to 69% . Our optimizer never increases the number of sensor reads or the amount of transferred data. We allows for prioritizing queries by penalizing read time deviations. We show two examples with gentle and strong prioritizations.

We require read time tolerances to enable frequent read and traffic sharing among queries. Our experiments show that AdaM and FAST, as examples for adaptive sampling techniques, are robust against read time slack. This verifies that read time tolerances are applicable to adaptive sampling techniques.

8 RELATED WORK

The problem of oversampling has been studied from various angles. However, we observed that there is no *one-fits-all* solution: either algorithms are limited to specific use-cases [17, 33], miss adaptivity [26, 38], or do not consider shared sensors [36, 41]. In the following section, we discuss how *UDSFs* and *multi-query read scheduling* incorporate, extend, and aid existing oversampling reduction techniques. We then present related work from the field of sensor networks regarding multi-query optimization and sensor read scheduling.

Reduced Oversampling. TinyDB [26] introduces the concept of acquisitional query processing (ACQP) to control when and how often to sample. It arranges database operators together with fetch operators (sensor reads) in a common processing pipeline. However, TinyDB only allows for periodic sampling algorithms at the source of a processing pipeline. This still leads to oversampling because it prevents adaptive sampling [17, 19, 41]. Our UDSFs overcome this limitation. Our sensor read scheduler further complements TinyDB with the ability for read and traffic sharing.

Model-driven data acquisition is another way to reduce the number of sensor reads [10, 15, 22, 33, 42]. One can implement model-driven data acquisition as UDSF as we have shown in Section 5.5. The proposed algorithms neither mention nor hinder read sharing among queries.

An orthogonal approach to reduce oversampling is the joint optimization of data acquisition and delivery [36]. This method trades-off data transfer costs (slow transfer, low cost) against sampling costs (high frequency, high cost) while providing data freshness guarantees. Our work complements this approach because the data freshness benefits from read and traffic sharing among queries.

Multi Query Optimization. Several works propose to optimize the query execution across queries and users in sensor networks [25, 46]. However, these publications do not consider sensor read scheduling and can be applied supplementary to our scheduling algorithm.

Xiang et al. [44, 45] as well as Mueller and Alonso [30] optimize a batch of queries as a whole to eliminate redundancies and fuse similar operations. They set the sampling rate of sensors to be the greatest common divisor (GCD) of the sampling rates from all queries. In contrast to our work, both approaches rely on periodic sampling to compute the required GCD.

Scheduling Algorithms. Tavakoli et al. [38] also utilize read time tolerances for sensor read scheduling. They model overlaps of tolerance intervals in an online evolving interval-cover graph which they use to determine read times. In contrast to our solution, their approach is limited to periodic sampling and does not not optimize exact read times.

Fang et al. [18] and CATS [47] address the issue of sampling continuous intervals (e.g. video and audio recording). They explore tolerances in the placement of recording intervals to maximize interval overlaps. We consider the challenge to maximize interval overlap as orthogonal to the optimization of exact sensor read times. Further scheduling algorithms from the field of sensor networks study transmission scheduling [5, 16, 29, 40]. They switch between sleep times and transmission periods in order to save energy, but do not cover sensor read scheduling or read sharing among queries.

In summary, our *UDSFs* and *multi-query read scheduling* form a common framework to incorporate the presented oversampling reduction techniques. *UDSFs* work as general abstraction for sampling functions. *Multi-query read scheduling* transparently multiplexes *UDSFs* on shared sensors, leading to a global cost optimization.

9 CONCLUSION

We introduce user-defined sampling functions (UDSFs) as well as a multi-query scheduling algorithm for sensor reads. These are powerful means to solve the problem of *oversampling*: USDFs enable diverse adaptive sampling techniques and allow for defining the data demand of each query. Our multi-query scheduling algorithm multiplexes UDSFs and utilizes read time tolerances to minimize sensor reads with respect to query needs. The complexity of multi-query scheduling is transparent to the user. Our experimental evaluations show savings of 87% in sensor reads and data transfers for an example with real-world sensor data. In addition,

our read time optimizer reduces the deviation from desired read times by up to 69% in our experiments. We further allow for prioritizing queries in a flexible way. Overall, on-demand data streaming from sensor nodes leads to significantly reduced sampling rates and corresponding savings in communication costs.

Acknowledgements. We thank Julius Hülsmann, Vianney de Cibeins, and Philipp Grulich for their assistance.

This work received funding through the EU Horizon 2020 projects Streamline (688191) and SAGE (671500) and from the German Ministry for Education and Research as Berlin Big Data Center (01IS14013A) and Software Campus (01IS12056).

REFERENCES

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al. 2014. The Stratosphere platform for big data analytics. *VLDB Journal* 23, 6 (2014), 939–964.

[2] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. 2013. Photon: Fault-tolerant and scalable joining of continuous data streams. ACM SIGMOD, 577–588.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL continuous query language: semantic foundations and query execution. *VLDB Journal* 15, 2 (2006), 121–142.

[4] Martin Arlitt, Manish Marwah, Gowtham Bellala, Amip Shah, Jeff Healey, and Ben Vandiver. 2015. IoTAbench: an internet of things analytics benchmark. ACM - International Conference on Performance Engineering, 133–144.

[5] Andreea Berfield and Daniel Mossé. 2006. Efficient scheduling for sensor networks. IEEE International Conference on Mobile and Ubiquitous Systems, 1–8.

[6] E Brockmeyer, HL Halstrøm, Arne Jensen, and Agner Krarup Erlang. 1948. The life and works of A. K. Erlang. (1948).

[7] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin* 38, 4 (2015).

[8] Paris Carbone, Jonas Traub, Asterios Katsifodimos, Seif Haridi, and Volker Markl. 2016. Cutty: Aggregate Sharing for User-Defined Windows. ACM - Conference on Information and Knowledge Management.

[9] Kevin Carter and William Streilein. 2012. Probabilistic reasoning for streaming anomaly detection. IEEE Statistical Signal Processing Workshop, 377–380.

[10] David Chu, Amol Deshpande, Joseph M Hellerstein, and Wei Hong. 2006. Approximate data collection in sensor networks using probabilistic models. IEEE International Conference on Data Engineering, 48–48.

[11] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. 2003. Approximate join processing over data streams. SIGMOD, 40–51.

[12] Anirban DasGupta. 2011. Poisson Processes and Applications. In *Probability for Statistics and Machine Learning: Fundamentals and Advanced Topics*. Springer New York, 437–462.

[13] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[14] Alan Demers, Johannes Gehrke, Rajmohan Rajaraman, Niki Trigoni, and Yong Yao. 2003. The cougar project: a work-in-progress report. *Sigmod Record* (2003).

[15] Amol Deshpande, Carlos Guestrin, Samuel R Madden, Joseph M Hellerstein, and Wei Hong. 2004. Model-driven data acquisition in sensor networks. International Conference on Very Large Data Bases, 588–599.

[16] Partha Dutta, Vivek Mhatre, Debmalya Panigrahi, and Rajeev Rastogi. 2010. Joint routing and scheduling in multi-hop wireless networks with directional antennas. IEEE International Conference on Computer Communications, 1–5.

[17] Liyue Fan and Li Xiong. 2014. An adaptive approach to real-time aggregate monitoring with differential privacy. *IEEE Transactions on Knowledge and Data Engineering* 26, 9 (2014), 2094–2106.

[18] Xiaolin Fang, Hong Gao, Jianzhong Li, and Yingshu Li. 2013. Application-aware data collection in Wireless Sensor Networks. IEEE International Conference on Computer Communications, 1645–1653.

[19] Elena I Gaura, James Brusey, Michael Allen, Ross Wilkins, Dan Goldsmith, and Ramona Rednic. 2013. Edge mining the internet of things. *IEEE Sensors Journal* 13, 10 (2013), 3816–3825.

[20] Google. 2017. Cloud Prediction API - Pricing and Terms of Service, accessed 05.05.17. Prices: $0.50/1,000 predictions beyond the initial 10,000. (2017). https://cloud.google.com/prediction/pricing.

[21] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, et al. 2013. IBM streams processing language: Analyzing big data in motion. *IBM Journal* 57, 3/4 (2013), 7.1–7.11.

[22] Ankur Jain, Edward Y Chang, and Yuan-Fang Wang. 2004. Adaptive stream resource management using kalman filters. SIGMOD, 11–22.

[23] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. 2006. On-the-fly sharing for streamed aggregation. SIGMOD, 623–634.

[24] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record* 34, 1 (2005), 39–44.

[25] Ming Li, Tingxin Yan, Deepak Ganesan, Eric Lyons, Prashant Shenoy, Arun Venkataramani, and Michael Zink. 2007. Multi-user data sharing in radar sensor networks. ACM Conference on Embedded Networked Sensor Systems, 247–260.

[26] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. 2005. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems* 30, 1 (2005), 122–173.

[27] Gianmarco De Francisci Morales and Aristides Gionis. 2016. Streaming similarity self-join, Vol. 9. International Conference on Very Large Data Bases, 792–803.

[28] Kannan M Moudgalya. 2007. Proportional, Integral, Derivative Controllers. *Digital Control, Wiley* (2007), 301–325.

[29] Conor Muldoon, Niki Trigoni, and Greg MP O'Hare. 2011. Combining sensor selection with routing and scheduling in wireless sensor networks. International Workshop on Data Management for Sensor Networks.

[30] Rene Muller and Gustavo Alonso. 2006. Efficient sharing of sensor networks. IEEE Mobile Adhoc and Sensor Systems, 109–118.

[31] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. 2013. The DEBS 2013 grand challenge. ACM International Conference on Distributed Event-Based Systems, 289–294.

[32] Valentin Protschky, Christian Ruhhammer, and Stefan Feit. 2015. Learning traffic light parameters with floating car data. In *IEEE Intelligent Transportation Systems (ITSC)*. IEEE, 2438–2443.

[33] Usman Raza, Alessandro Camerra, Amy L Murphy, Themis Palpanas, and Gian Pietro Picco. 2012. What does model-driven data acquisition really achieve in wireless sensor networks? IEEE International Conference on Pervasive Computing and Communications, 85–94.

[34] Felix Rempe, Philipp Franeck, Ulrich Fastenrath, and Klaus Bogenberger. 2016. Online Freeway Traffic Estimation with Real Floating Car Data. In *IEEE Intelligent Transportation Systems (ITSC)*. IEEE, 1838–1843.

[35] A Wayne Roberts and Dale E Varberg. 1973. *Convex functions*. Academic Press.

[36] Antonios Skordylis and Niki Trigoni. 2009. Jointly optimizing data acquisition and delivery in traffic monitoring VANETs. ACM SIGAPP Symposium On Applied Computing, 2186–2190.

[37] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. International Conference on Very Large Data Bases, 309–320.

[38] Arsalan Tavakoli, Aman Kansal, and Suman Nath. 2010. On-line sensing task optimization for shared sensors. ACM/IEEE International Conference on Information Processing in Sensor Networks, 47–57.

[39] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@twitter. SIGMOD, 147–156.

[40] Niki Trigoni, Yong Yao, Alan Demers, Johannes Gehrke, and Rajmohan Rajaraman. 2007. Wave scheduling and routing in sensor networks. *ACM Transactions on Sensor Networks* 3, 1 (2007).

[41] Demetris Trihinas, George Pallis, and Marios D Dikaiakos. 2015. AdaM: An adaptive monitoring framework for sampling and filtering on IoT devices. IEEE Big Data.

[42] Daniela Tulone and Samuel Madden. 2006. PAQ: Time series forecasting for approximate query answering in sensor networks. European Workshop on Wireless Sensor Networks, 21–37.

[43] Rob van der Meulen. 2015. Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015. *Gartner Newsroom Press Release* (2015).

[44] Shili Xiang, Hock Beng Lim, and Kian-Lee Tan. 2006. Impact of multi-query optimization in sensor networks. International Workshop on Data Management for Sensor Networks, 7–12.

[45] Shili Xiang, Hock Beng Lim, Kian-Lee Tan, and Yongluan Zhou. 2007. Two-tier multiple query optimization for sensor networks. IEEE International Conference on Distributed Computing Systems, 39–39.

[46] Shili Xiang, Wei Wu, and Kian-Lee Tan. 2012. Optimizing Multiple Data Acquisition Queries in Sparse Mobile Sensor Networks. IEEE International Conference on Mobile Data Management, 137–146.

[47] Yawei Zhao, Deke Guo, Jia Xu, Pin Lv, Tao Chen, and Jianping Yin. 2016. CATS: Cooperative Allocation of Tasks and Scheduling of Sampling Intervals for Maximizing Data Sharing in WSNs. *ACM TOSN* 12, 4 (2016), 29.