# Grand Challenge: Real-time Detection of Smart Meter Events with Odysseus

Michael Brand
michael.brand@uol.de
University of Oldenburg
Oldenburg, Germany

Tobias Brandt
tobias.brandt@offis.de
OFFIS—Institute for Information Technology
Oldenburg, Germany

Marco Grawunder
marco.grawunder@uol.de
University of Oldenburg
Oldenburg, Germany

## ABSTRACT

The energy grid is changing rapidly to include volatile, renewable energy sources to help achieve climate goals. The transition to a smart grid, including smart meters for the metering and communication of the energy consumption, helps with that transition. The smart meters provide a stream of measurements, which can be used for additional services, such as visualization of power consumption. Detecting switching events, when devices in a household are switched on or off, is one possible application on smart meter data.

The goal of the ACM DEBS Grand Challenge 2020 is to implement a live switch detection on a data stream from a smart meter for Non-Intrusive Load Monitoring. This paper presents a solution for the challenge with a general purpose and open source data stream management system that focuses on reusable, generic operators instead of a custom black-box implementation.

## CCS CONCEPTS

• **Information systems** → **Data streams**; **Stream management**; **Data stream mining**; • **Hardware** → *Smart grid*.

## KEYWORDS

Data Stream Management Systems, Data streaming, Stream management, Smart grid, Smart meter, Non-Intrusive Load Monitoring

**ACM Reference Format:**
Michael Brand, Tobias Brandt, and Marco Grawunder. 2020. Grand Challenge: Real-time Detection of Smart Meter Events with Odysseus. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3401025.3401757

## 1 INTRODUCTION

Smart meters are connected electricity meters. As part of the transition to a smart grid, smart meters allow to measure and analyze the power consumption with a high granularity and are, therefore, enablers for additional services, for example for customers to understand their power consumption.

Non-Intrusive Load Monitoring (NILM) is a technique to analyze the raw power consumption data to gain more detailed insights, e. g., when and which devices are switched on or off [2]. The ACM DEBS Grand Challenge 2020 (GC 2020) [3] provides a challenge to apply a given NILM algorithm on a stream of voltage and current that resembles the consumption of an office building. The algorithm aims to detect switching points, i. e., sudden changes in power consumption. It has to be applied on two data streams: one in which all data elements arrive in their original order and one in which neither the order nor the completeness of the data is guaranteed [3]. These two different input streams are referred to as Query 1 and Query 2. The goals for Query 1 are a low latency and a low total runtime while Query 2 aims at having both a high timeliness and a high accuracy. Reading fewer input elements before creating an output results in a high timeliness. Accuracy is defined by the difference in the event time stamps to the ones of the baseline solution, i. e., without missing or out-of-order elements.

The main idea of our solution is to not use black-box implementations of the algorithm, but to solve the problem with a general purpose data stream management system (DSMS) and its application-independent operators and data stream query language. We designed the queries based on existing functionality in the open source DSMS framework Odysseus[1] [1]. Odysseus is a flexible framework to create custom DSMSs based on generic feature sets such as relational algebra, key value processing, user management, query management and optimization, query languages, and access interfaces for different data sources such as http, files, MQTT, etc.

The main contributions described in this paper are the following:

- A solution for the GC 2020 with generic, application-independent, and most of them basic operators.
- Query plans that can easily be adapted or used for similar problems by changing parameters.

The paper continues in Section 2 with the description of the event detection. Next, in Section 3, different possibilities to handle missing and out-of-order elements are described and discussed. Section 4 presents the evaluation results and Section 5 concludes the paper.

## 2 EVENT DETECTION

The challenge to solve is to detect events, the turning on or off of a device, in a data stream of a smart meter. The input data stream consists of energy measurement tuples, each providing the voltage $v$ and the current $c$ for a point in time. For the event detection, a specific algorithm is mandated. The algorithm is a two-stage clustering algorithm based on DBScan with a forward and a backward pass [2]. Its input features are active and reactive power data that need to be calculated from the smart meter input data. The algorithm uses

[1]https://odysseus.uni-oldenburg.de/

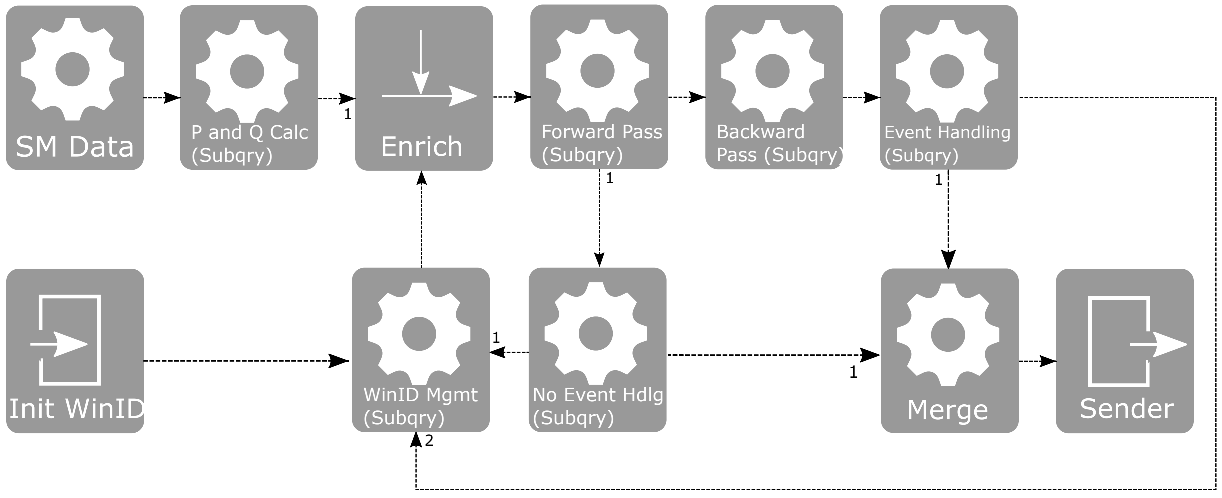


**Figure 1: Query plan of the event detection on an abstract level (Query 1). Subquery operators contain other query plans.**

a tuple-based window, which size increases with every new input feature pair. It is cleared if an event is detected or if its size exceeds 100 elements. In the forward pass, the DBScan algorithm is applied to the window and it is checked whether an event occurs in the window or not. The latter depends on how events are modeled. Barsim and Yang [2] propose three different event models $\mathcal{M}_1$, $\mathcal{M}_2$, and $\mathcal{M}_3$. The event models differ in the constraints for having an event in the current window [3]. $\mathcal{M}_3$ is used in this paper because it is the most general one.

If an event is detected in the forward pass, the window is forwarded to the backward pass to improve the localisation of the event. For this, the oldest feature pairs are iteratively removed from the window before DBScan and the model check is applied again. This is done until the event is not detected any more. Then, the solution from the previous iteration, i. e., the last iteration that found the event, is chosen. This backward pass leads to more stable selections.

The query plan of our solution is visualized in Figure 1 and contains several logical parts implemented as subqueries. Subquery operators are special operators in Odysseus that hide nested query plans. The "feature calculation", for example, hides a query plan that calculates the active and reactive power based on the smart meter input data. Subqueries are normal queries. Hence, they are installed and started together with the main query. The benefits of subqueries are, on the one hand, a higher clarity due to several layers of abstraction and, on the other hand, a better exchangeability. The latter is achieved because different queries can be defined for the same (partial) problem and be easily exchanged by just calling a different subquery. We demonstrate this feature in Section 3 by proposing different subqueries for the out-of-order handling.

Another general feature of the query plan to be mentioned is the possibility to design recursive plans. This is a new feature of Odysseus that we developed to solve the GC 2020. It is also a good example of how we use challenges like the GC 2020 to improve Odysseus by adding application-independent, useful mechanisms and operators.

In the following, the query plan will be explained. In the "feature calculation", the single energy measurements are aggregated using a tuple-based window with a size and an advance of 1000 to calculate the active and reactive power ($P$ and $Q$) for periods in time. The active power is defined as

$$P = \frac{1}{size(W1)} \cdot \sum (v_i \cdot c_i)$$

for all $v_i$ and $c_i$ in the same window. To calculate the reactive power, the apparent power $S$ must be calculated first:

$$S = v_{RMS} \cdot c_{RMS}$$

with $v_{RMS}$ and $c_{RMS}$ being the root-mean-squares (RMSs) of all $v_i$ and $c_i$ in the same window. The reactive power is then defined as

$$Q = \sqrt{S^2 - P^2}.$$

The "window ID management" manages the current window ID for the dynamic window of the forward pass. The window ID is an increasing sequence number, with which every input tuple is enriched afterward. The window ID is needed to determine whether the current window must be extended or cleared. The source named "initial window ID" provides the initial sequence number and, after each clearing of the window, the window ID increases.

The "forward pass" subquery uses the DBScan algorithm in a map operator to perform the clustering. A model check afterward determines whether an event occurs in the current window or not, based on $\mathcal{M}_3$ from Barsim and Yang [2], which is slightly adapted in the GC 2020. Figure 2 shows the query plan of the model check subquery.

The first constraint of $\mathcal{M}_3$ is that the clustering result must contain at least two non-outlier clusters [2]. In addition, for the GC 2020, we defined in our solution that the outlier cluster must be not empty. The second constraint is that there must be at least two non-outlier clusters with a high temporal locality. The temporal locality of a cluster $C_i$ is defined in [2] as

$$Loc(C_i) = \frac{|C_i|}{Len(C_i)}.$$

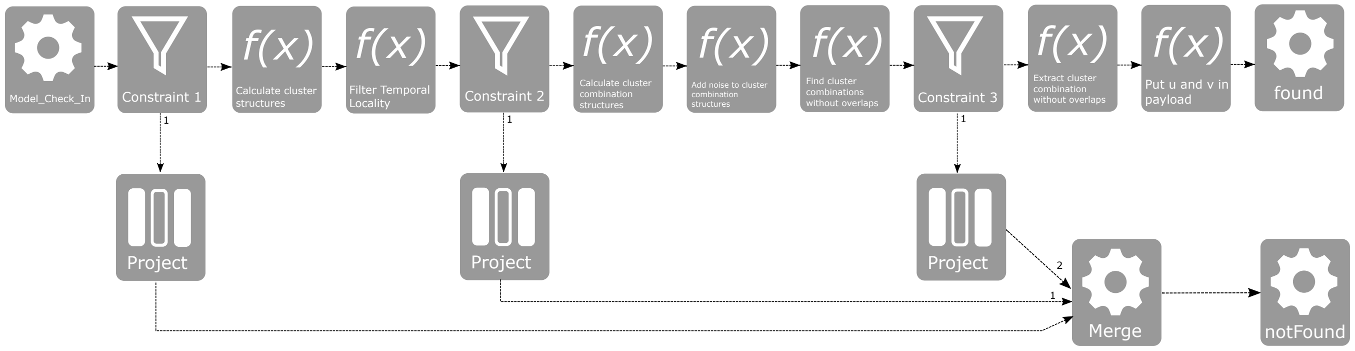

Model_Check_In
Constraint 1
f(x)
Calculate cluster structures
f(x)
Filter Temporal Locality
Constraint 2
f(x)
Calculate cluster combination structures
f(x)
Add noise to cluster combination structures
f(x)
Find cluster combinations without overlaps
Constraint 3
f(x)
Extract cluster combination without overlaps
f(x)
Put u and v in payload
found
1
1
1
Project
Project
Project
2
1
Merge
notFound

**Figure 2: The subquery to locate events in a window based on the DBScan output.**

It is the proportion of the amount of feature pairs in the cluster and the temporal length of the cluster. The latter is the difference in the time stamps of the feature pair with the highest time stamp and the one with the lowest.

In the query plan in Figure 2, cluster structures are calculated before the second constraint is checked. A cluster structure is a list of the size of the cluster, the lowest as well as the highest time stamp in the cluster. All other information such as the concrete feature pairs in the cluster are discarded because they are not needed anymore after the clustering. Of course, the feature pairs are needed again for the clustering in the backward pass. For that, the content of the current window is stored in a separate attribute, which is not touched in the model check.

The third constraint is that there must be at least two non-outlier clusters that do not overlap in the time domain [2]:

$$\exists u, v > u : \boldsymbol{x}_u \notin C_1 \forall n > u \land \boldsymbol{x}_u \in C_1 \land \boldsymbol{x}_v \notin C_2 \forall n < v \land \boldsymbol{x}_v \in C_2.$$

$\boldsymbol{x}_n$, $\boldsymbol{x}_u$, and $\boldsymbol{x}_v$ are feature pairs with the time stamp $i$, $u$, and $v$, respectively. $C_1$ is the left cluster (before the event) and $C_2$ the right (after the event). In addition, for the GC 2020, we defined in our solution that at least one outlier feature (cluster $C_0$) must be in the so-called change interval between $C_1$ and $C_2$:

$$\exists \boldsymbol{x}_n \in C_0 : u < n < v.$$

We use different map operations to preprocess the third constraint. Listing 1 shows the preprocessing in the query language PQL[2]. In a first map operation, all possible cluster combination structures are calculated and stored in a list attribute. A cluster combination structure consists of a left cluster structure, a right cluster structure, the end of the left cluster (highest time stamp, named $u$), and the start of the right cluster (lowest time stamp, named $v$). The `#DEFINE` commands are string replacements to improve readability. In a second map operation, we add the time stamps of all outliers that lie between $u$ and $v$ to each cluster combination structure. The cluster combination structures are filtered by the constraint that $u$ must be before $v$ and the list of outliers between $u$ and $v$ must be not empty. This is done in a third map operation. The validation of the third constraint is then to check whether at least one cluster combination structure is left in the list.

[2]https://wiki.odysseus.informatik.uni-oldenburg.de/x/NQFG

If an event is found, the start and the end of the change interval, i. e. the lowest and highest time stamps of the outliers, are stored in an attribute called payload. This is useful for the backward pass because the payload attribute is only changed when an event is detected. If there is no event detected in an iteration of the backward pass, the best fitting change interval from the last iteration is still in the payload. The windows in which no event is detected (output port 1 of the filters in Figure 2), are forwarded to another output of the subquery.

The found events are forwarded to the backward pass (cf. Figure 1). The differences between the forward and the backward pass are the following:

- The backward pass does not have a window operator.
- The window forwarded from the forward pass is sorted according to the timestamps.
- The backward pass contains a recursive plan after the sorting. The recursion finishes when an event is not detected any more.
- Before the clustering in the backward pass, the feature pair with the oldest time stamp is removed from the window.

The backward pass has, in difference to the forward pass, only one output channel. The found events are forwarded to the "event handling" subquery via this output channel. The "event handling" subquery has two purposes. First, the found events are formatted according to the desired format of the results. Second, the point in time of the end of the found change interval is forwarded to the "window ID management" to increase the window ID and to remove all elements till the forwarded point in time from the window.

Analogous, windows in which no event is detected in the forward pass are managed in the "no event handling" subquery. The "no event detection" subquery has two purposes, too. First, the information that there is no event in the window is formatted according to the desired format of the results. Second, if the window contains more than 100 feature pairs, a trigger is sent to the "window ID management" to increase the window ID and to remove all elements from the window.

## 3 OUT-OF-ORDER HANDLING

The second part of the challenge is to handle smart meter data for the event detection that can be, regarding their time stamps, out-of-order or even missing [3]. Figure 3 shows the query plan for

**Listing 1: Preprocessing for the third model constraint.**

```
/// first build cluster combinations: [left cluster, right cluster, u, v]
#DEFINE c_left eif(elementAt(asList(c1),1) < elementAt(asList(c2),1),c1,c2) /// cl. with lower min
#DEFINE c_right eif(elementAt(asList(c1),1) < elementAt(asList(c2),1),c2,c1) /// other cluster
#DEFINE u elementAt(asList(${c_left}), 2) /// u is max of left cluster
#DEFINE v elementAt(asList(${c_right}), 1) /// v is min of right cluster
combinations = MAP({
                expressions = [
                  ['forEachPair(cluster_structures,"toList(${c_left},${c_right},${u},${v})")', '
                      ↪ cluster_combination_structures', 'List_List'],
                  'noise_cluster', 'payload', 'window'
                ]
              },
              0:constraint_2
            )
#DEFINE u elementAt(asList(cluster_combination_structure), 2)
#DEFINE v elementAt(asList(cluster_combination_structure), 3)
#DEFINE noise filter(asList(noise_cluster),toTuple(cluster_combination_structure),\'a_noise>${u}&&
    ↪ a_noise<${v}\') /// noise between u and v
add_noise = MAP({
                expressions = [
                  ['forEach(cluster_combination_structures,toTuple(noise_cluster),"addTo(${noise},
                      ↪ cluster_combination_structure)")', 'cluster_combination_structures', '
                      ↪ List_List'],
                  'payload', 'window'
                ]
              },
              combinations
            )
#DEFINE noise asList(elementAt(asList(cluster_combination_structure), 4))
without_overlaps = MAP({
                    expressions = [
                      ['filter(cluster_combination_structures,"${u}<${v}&&!isEmpty(${noise})")', '
                          ↪ cluster_combination_structures', 'List_List'],
                      'payload', 'window'
                    ]
                  },
                  add_noise
                )
```

Query 2. The query is almost the same as the first (cf. Figure 1) but contains an "out-of-order management" subquery previous to the calculation of the features.

Missing elements are handled by not using an element window for $W1$ but to use a predicate window[3]. An element window would collect, in case of the GC 2020, exactly 1000 elements. Hence, missing elements would shift the placement of elements to the windows. With the predicate window operator, a window is complete if the attribute $i$ of the current element (the time stamp) is a multiple of 1000: $i\%1000 == 0$. The use of this condition results in the same window arrangement as without missing elements.

In the remainder of this section, we provide two different methods to manage out-of-order elements. The first method is to discard elements that belong, with respect to its $i$, to a window that is already processed. We call this method *simple out of order* (*Simple OoO*). The idea, besides the discarding, is that elements only need to be discarded if they would be allocated to the wrong window $W1$ for the calculation of the features. Within a window, the order of the elements is irrelevant (cf. calculation of $P$ and $Q$ in Section 2). The time stamps ($i$) of the input data for the GC 2020 start with 0 and the size of $W1$ is 1000. Then, the condition for discarding an element is

$$\left\lfloor \frac{last\_i}{size(W_1)} \right\rfloor > \left\lfloor \frac{i}{size(W_1)} \right\rfloor$$

with $last\_i$ being $i$ of the last element before the current one.

The second method is to use a reorder operator[4] in combination with heartbeats. A heartbeat operator sends heartbeats to the

[3]https://wiki.odysseus.informatik.uni-oldenburg.de/x/JICO

[4]https://wiki.odysseus.informatik.uni-oldenburg.de/x/24Fs

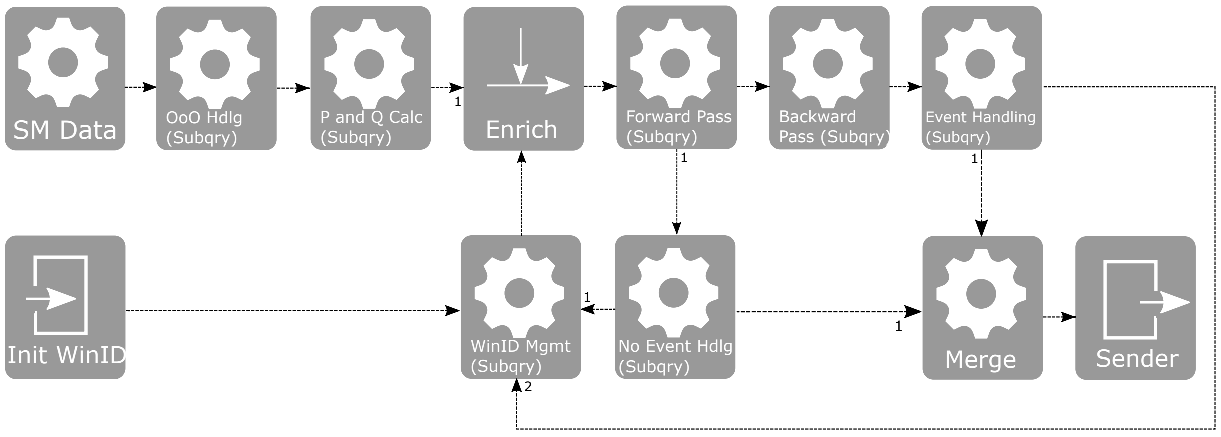


**Figure 3: Query plan of the event detection and handling of out-of-order elements on an abstract level (Query 2). Subquery operators contain other query plans.**

reorder operator for incoming elements. The calculation for the timestamp of the heartbeat can be defined by the user. The reorder operator collects all incoming elements and sorts them. When it receives a heartbeat, it sends all elements for which time stamp is before the time stamp of the heartbeat in the correct order out to the next operator. The parameter to adjust for the use case of the GC 2020 is the calculation for the timestamp of the heartbeat:

$$\lfloor i - r \cdot size(W_1) \rfloor - 1$$

with a reorder factor $r$. In other words, $r$ is the amount of window batches in the past, which elements still need to be kept by the reorder operator, when it receives the heartbeat. If $r = 0$ is chosen, all elements of older window batches are sent by the reorder operator for further reordering. We evaluated this method with different values for $r$ to find good tradeoffs between a high waiting time (low timeliness) and a high accuracy of the results.

We implemented the methods in different "out-of-order management" subqueries. The results are discussed in Section 4.

## 4 EVALUATION

The GC 2020 provides two sets of data to test and evaluate the solutions. The larger data set contains 15 000 000 tuples. The following evaluations have been conducted on this data set with a benchmarking software provided by the GC 2020. Additionally, the challenge includes a public evaluation platform to benchmark the submitted solutions against each other. The benchmarking software measures four aspects of the respective solution, two for each query. The two aspects for Query 1 are:

- **Total Runtime:** The total amount of time that the query needs to process on the provided benchmarking system. Lower is better.
- **Latency:** The average amount of time that the solution needs to create a result for a new input. Lower is better.

The measured values for Query 2 are:

- **Timeliness:** The basis for the timelineness is the time stamp of the last read input element when an output is created. The timelineness compares the value of the provided solution ($t'_{in}$) with the value of the reference solution ($t_{in}$) for each output: $\sum \max(0, 1 - \frac{t'_{in} - t_{in}}{10})$. Higher is better.
- **Accuracy:** The basis for the accuracy is the time stamp of found events. The accuracy compares the value of the provided solution ($t'_{out}.events_s$) with the value of the reference solution ($t_{out}.events_s$) for each event: $\sum \max(0, 1 - \frac{|t'_{out}.events_s - t_{out}.events_s|}{10})$. Higher is better.

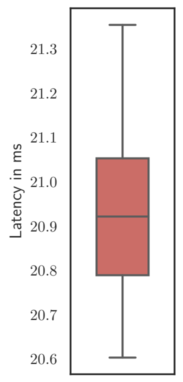
These measured values are used to rank the solutions on the benchmarking system. Additionally, we conducted benchmarks on our own machines with the same benchmarking software and the provided data set to measure the statistical spread in a controlled environment. The benchmarks were conducted on an Ubuntu 19.10 system with Java 11, 16 GiB of RAM from which Odysseus was able to use 8 GiB. The machine runs on an Intel i5-6200U CPU with 2.30 GHz.

For Query 1 we have exactly one solution for which we measured the latency and the total runtime. The results can be seen in Figure 4. As can be seen, the latency is about 21 ms. The benchmarking software sends the data in batches with 1 000 tuples at a time, wherefore the latency is the time from receiving such a batch until producing the output for the batch.

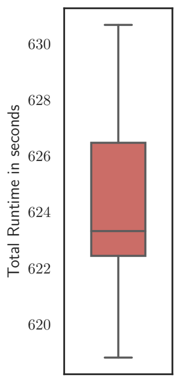
As can be seen in Figure 4b, the whole query needs about 10.5 minutes to finish. As the original data rate is not known, we cannot say whether this is slower or faster than real time.

Query 2 benchmarks the out-of-order solutions with an input data stream with missing and out-of-order values. We measured the accuracy and timeliness for both the *Simple OoO* and the *Reorder* algorithms. For the *Reorder* algorithm, we evaluated different settings for the reorder factor parameter ($r$). The results can be seen in Figure 5 and Figure 6.

Figure 5 shows the *Reorder* algorithm with two extreme settings for the reorder factor and the *Simple OoO* algorithm. A reorder factor of -1 means that no reorder takes place and the operator does not wait at all while a reorder factor of 20 means that the algorithm waits for the $n + 20$th batch of elements before processing the $nth$ batch of elements.

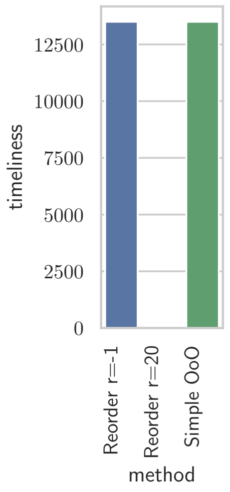


(a) Latency of Query 1

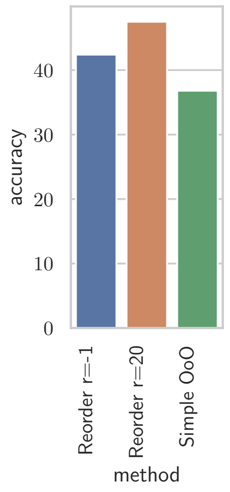


(b) Total runtime of Query 1

**Figure 4: The latency (4a) and total runtime (4b) results as boxplots of 14 runs for Query 1.**



(a) Timeliness of Query 2



(b) Accuracy of Query 2

**Figure 5: The timeliness (5a) and accuracy (5b) results of the different solutions for Query 2 compared.**

For the timeliness value, the *Reorder r=-1* surpasses the *Simple OoO* slightly with a value of 13 499 versus 13 497. The *Reorder r=20* has a timeliness of zero because it waits too long before producing the results. While this behavior leads to the worst timeliness, it is the solution with the highest accuracy with a value of 47.5. *Reorder* and *Simple OoO* follow with values of 42.4 and 36.8.

The benchmarks show that there is a tradeoff between the timeliness and the accuracy. Nevertheless, when only comparing *Reorder r=-1* and *Simple OoO*, *Reorder r=-1* performs better in both categories while the *Reorder r=20* solution is only an option in cases where the timeliness is not important.

To evaluate the impact of the reorder factor, we measured the accuracy and timeliness with an increasing $r$. The results are shown in Figure 6. The accuracy increases with an increasing value for $r$ while the timeliness drops down to zero with a reorder factor of 8.

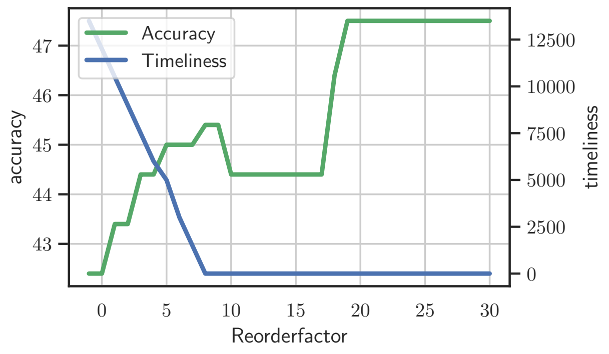


**Figure 6: Impact of the reorder factor on the benchmark results of Query 2.**

The advantage of a parameterized solution like the *Reorder* algorithm is that the parameters can be chosen based on the use case. Doing so, it is possible to find a sweet spot between accuracy and timeliness.

## 5 CONCLUSION

The GC 2020 [3] sets up a stream processing challenge with a real-world scenario in the field of smart meter data. Switch events that occur when a device is switched on or off have to be detected in a raw electricity consumption data stream with a specified algorithm.

Our presented solutions focus on using a general purpose DSMS, namely Odysseus, with operators that are not especially made for this specific challenge but that implement generic data stream processing functionality. This makes our solution simpler to understand and adapt due to a data stream query language and tool support, that, for example, shows the live stream results as well as the query graph.

We used these operators to implement the specified algorithm in two different ways. While the main query does not need to handle out-of-order elements, we adapted this query to handle those for Query 2. Due to the use of a data stream query language, the adaption of the query has been a minor change. Hence, the GC 2020 showed that data stream problems can be solved with general purpose systems without the need for problem-specific low-level implementations.

## REFERENCES

[1] H.-Jürgen Appelrath, Dennis Geesen, Marco Grawunder, Timo Michelsen, and Daniela Nicklas. 2012. Odysseus - A Highly Customizable Framework for Creating Efficient Event Stream Management Systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-based Systems*. Association for Computing Machinery, Berlin, Germany, 367–368. https://doi.org/10.1145/2335484.2335525

[2] Karim Said Barsim and Bin Yang. 2016. Sequential clustering-based event detection for non-intrusive load monitoring. *Computer Science & Information Technology* 6 (2016), 77–85.

[3] Vincenzo Gulisano, Daniel Jorde, Ruben Mayer, Hannaneh Najdataei, and Dimitris Palyvos-Giannas. 2020. The DEBS 2020 Grand Challenge. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems (DEBS '20)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3401025.3402684