Grand Challenge: Optimized Parallel Implementation of Sequential Clustering-Based Event Detection

Gengtao Xu
Boston University
Boston, MA, United States
gengtaox@bu.edu

Jing Qin
Boston University
Boston, MA, United States
qinjin@bu.edu

Runqi Tian
Boston University
Boston, MA, United States
rqtian@bu.edu

ABSTRACT

The ACM 2020 DEBS Grand Challenge focused on Non-Intrusive Load Monitoring (NILM). NILM is a method that analyzes changes in the voltage and current going into a building to deduce appliance use and energy consumption. The 2020 Grand Challenge requires high performance and high accuracy NILM implementations. In this paper, we describe the technical details of our solution for the 2020 Grand Challenge, a NILM program based on Apache Flink. We employ a Divide-and-conquer strategy to implement our parallel algorithm and designed a verify stage to improve the accuracy. For performance, our method achieves a great overall run time and the highest accuracy.

CCS CONCEPTS

• **Computing methodologies → Vector / streaming algorithms**.

KEYWORDS

data stream processing, NILM, parallelization

ACM Reference Format:
Gengtao Xu, Jing Qin, and Runqi Tian. 2020. Grand Challenge: Optimized Parallel Implementation of Sequential Clustering-Based Event Detection. 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/10.1145/3401025.3401760

1 INTRODUCTION

The ACM 2020 DEBS Grand Challenge[3] focused on Non-Intrusive Load Monitoring (NILM), which is a process for analyzing changes in the voltage and current going into a building and deducing what appliances are used in this building as well as their energy consumption[1].

The goal of the challenge is to implement a NILM program, with two queries as tasks to solve. The first query aims at detecting devices that are turned on or off in a stream of voltage and current readings from a smart meter, resembling the aggregated energy consumption in an office building. The second query is a variation of the first one, and it is expected to process an input stream that can contain both late arrivals and missing tuples [3].

DEBS '20, July 13–17, 2020, Virtual Event, QC, Canada

ACM ISBN 978-1-4503-8028-7/20/07…$15.00
https://doi.org/10.1145/3401025.3401760

This paper describes our design and implementation of the two Grand Challenge queries using the Apache Flink stream processor.

In reduce overall processing time and improve performance, the central idea of our solution is to implement an event detector that processes streamed data in parallel.

The detection task we implemented consists of the following processing stages:

- Each input tuple is first aggregated using a tuple-based window W_1 of size and advance 1000 to compute the active and reactive power features.
- The features are divided into partitions and processed by data-parallel tasks.
- In each partition, the features are processed by the specified algorithm[1] and each new pair of features, consisting of active and reactive power, is added to a second window, W_2. The DBSCAN[6] algorithm is then applied to this window.
- The results of all partitions are merged and verified, and then the result is sent to a data sink.

We firstly introduce the necessary preliminary concepts in Section 2. Then in Section 3 we describe the input dataset and present some basic analysis results on the dataset. Section 4 shows our solution details. We discuss the technical challenges we faced in Section 5, and conclusion as well as potential optimization in Section 6.

2 BACKGROUND

2.1 NILM and DBSCAN

Non-Intrusive Load Monitoring (NILM), determines the energy consumption of individual appliances turning on and off in an electric load, based on detailed analysis of the current and voltage of the total load, as measured at the interface to the power source is described [4]. DBSCAN is the clustering algorithm used in the Barsim paper, for each new input or backward pass, the Barism paper used DBSCAN to update the clustering structure of data and try detecting the event [1].

2.2 Apache Flink

In our implementation, we build a program running on Apache Flink. Apache Flink[2] is a framework and distributed processing engine for stateful computations over unbounded and bounded data streams. Flink has been designed to run in all common cluster environments, perform computations at in-memory speed and any scale. Flink provides many powerful stream processing features and enables easy data-parallel computation.

Flink provides a really simple programming model, facilitating our programming tasks. Flink will take care of the big data problem, such as machine communication, fault tolerance, reliability,

networking, etc. As a programmer, we just need to focus on the data processing logic. Programs are constructed as a series of transformations on streams and operators are connected into a directed acyclic graph, the dataflow plan. Each operator can be run by one or more tasks in parallel and Flink handles data partitioning, communication, and deployment. Operators can be simple event-at-a-time transformations, such as maps and filters, windows that discretize a stream into time intervals, or custom processing functions that can maintain state and set timers that trigger in future moments.

Another useful feature that we leverage is Flink's support for event time and watermarks. This mechanism can be used to handle out-of-order data streams, like the events provided as input to Query 2. Flink's event time represents the logical time when an event in the stream happened. Event time is based on timestamps that are attached to events before they enter the processing pipeline. Watermarks are generated by Flink sources and pushed in the stream to inform the system about the current event time and let operators manage the delayed data. A watermark is a global progress metric that indicates the point in time when we are confident that no more delayed events will arrive. When a Flink operator receives a watermark with time T, it can assume that no further events with timestamp less than T will be received [5].

3 A GLANCE AT THE DATASET

The data provided consists of energy measurements from a smart meter. The schema of the input tuple is thus $< i, v, c >$, where attributes i, v, and c represent the tuple sequence id, the voltage, and the current, respectively. Every 1000 input tuple corresponds to the readings of the smart meter for 1 second (1000 milliseconds). As the data is created continuously, batch processing or storing data is impractical. Hence, we use a stream processing platform to implement our solution. We make a preliminary inspection of the data and try to analyze its basic characteristics. As shown in Figure 1 and Figure 2, we select samples of each second from the 1st to the 5th second's reading and the 30th to the 60th second's reading every 10 seconds. We find that the voltage value is periodic with a tiny phase difference every second. We find that the current value exposes a similar behavior. According to these characteristics, we discuss how to deal with the missing data in section 5.4.

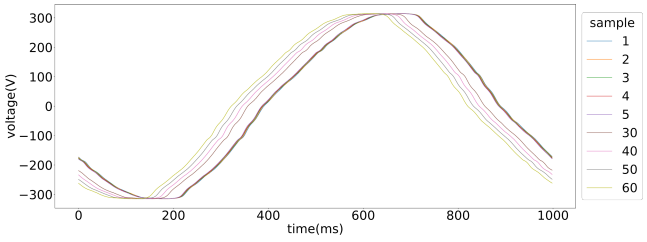

Figure 1: Voltage in each second's duration

4 SOLUTION ARCHITECTURE

4.1 Processing logic

Using a single-threaded program would result in processing incoming features one after the order in order, as described in the event

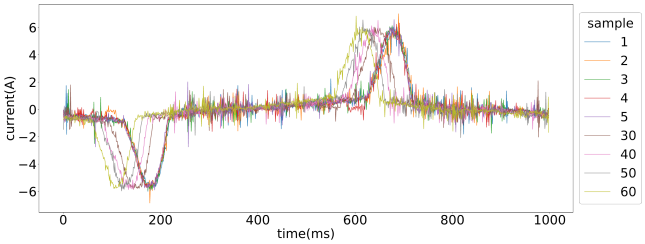

Figure 2: Current in each second's duration

detection algorithm introduced in the reference paper[1]. Such an approach could lead to bottlenecks if a part of the computation is too slow, and render the program unable to perform in real-time. For this reason, we decided to implement a parallel approach. The processing logic of our solution is shown in Figure 3.

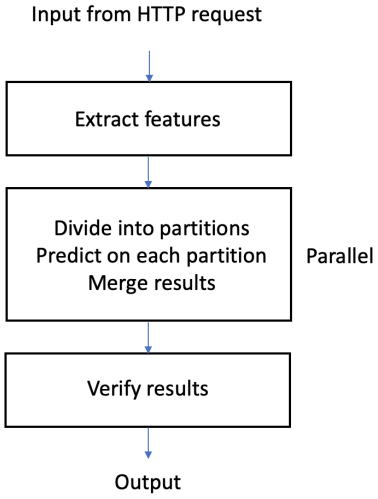

Figure 3: Processing logic

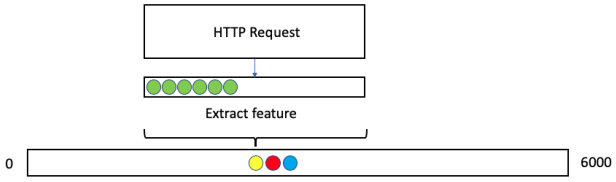

Figure 4: Extract features

4.2 Naive strategy

As described in the DEBS Challenge [3], the naive strategy for queries (Query 1 and Query 2) only process the input stream in a sequential manner. The processing time will be consumed a lot while computing the balanced event due to lots of time-consuming processes in event detector, such as updating cluster structure and

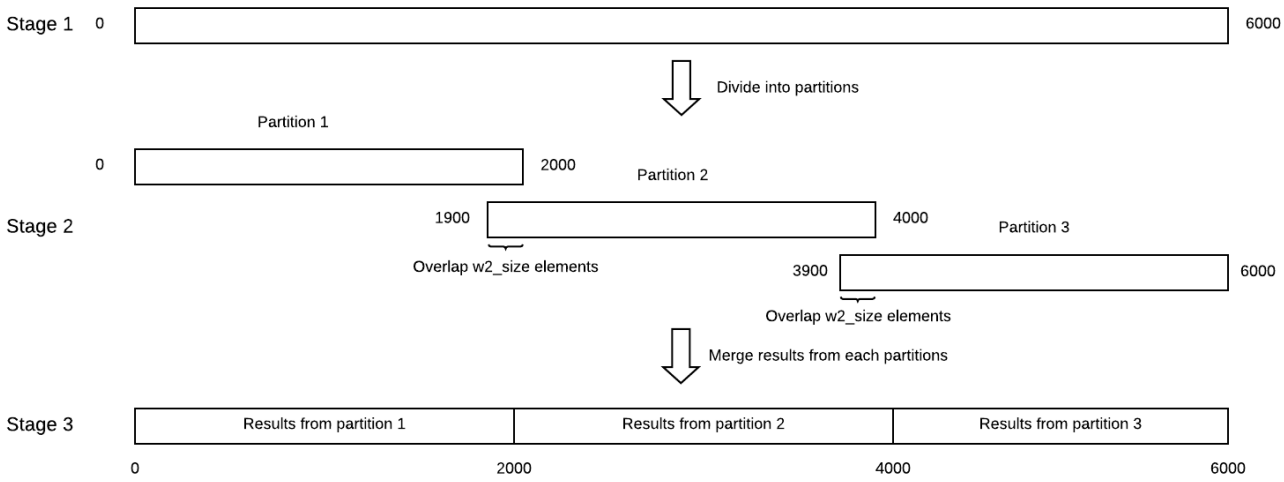

Figure 5: Parallel strategy

the whole backward pass, showed in Figure 9. Apparently, it is unnecessary to wait for each input result to process the next, hence, we describe our parallel strategy in the next section. Although processing the event stream in parallel is necessary, we did encounter many challenges while doing it. Since the Window2 position in the naive strategy is updated sequentially and it has high relevance with previous states, therefore, it is not easy to update it in a parallel fashion. Besides, the final results from the parallel method could be wrong due to the wrong Window2 start position. We will discuss how to fix these problems by introducing a verification process in section 4.4.

4.3 Parallel strategy

Our parallel strategy has the following requirements:

(1) It needs to be capable of scaling up or down to different numbers of parallel tasks.
(2) It needs to perform better than a single-threaded program.
(3) It needs to generate the same results as the single-threaded program.

We design a *divide-and-conquer* strategy to parallelize the NILM task. We introduce each step of the processing logic (Figure 3) in the rest of this section.

First, we pull data via an HTTP request and extract features from voltage and current values. This process is shown in Figure 4. Next, we perform the prediction on the features in parallel, as shown in Figure 5. Stage 1 represents the input features. We divide the input features into different partitions. Starting from the second partition, each subsequent partition needs to have an overlap of W_2 size (Window2 size) elements with the previous partition. This way, we can execute the prediction algorithm on each partition in parallel and generate results for each partition without waiting for the previous partition to finish processing. Then, we need to merge the partial results from each partition and generate the final results in Stage 3. For the overlapping part, we use the results from the previous partition. As a side-effect of leveraging parallelism, the order of original input changes, so we need to sort the results in Stage 3. This sorting stage runs sequentially on a single thread.

4.4 Verify merged result

A naive implementation of the parallel strategy described in the previous section might generate erroneous results. The naive implementation might fail to detect certain events, for example, the true value $< 2004, true, 2004 >$ is not detected. The naive implementation could also produce incorrect s values in detected events $< s, d, event\_s >$, for example, the true value should be $< 317, true, 314 >$ but the detected result is $< 319, true, 314 >$. We explain how to fix these two problems in this section.

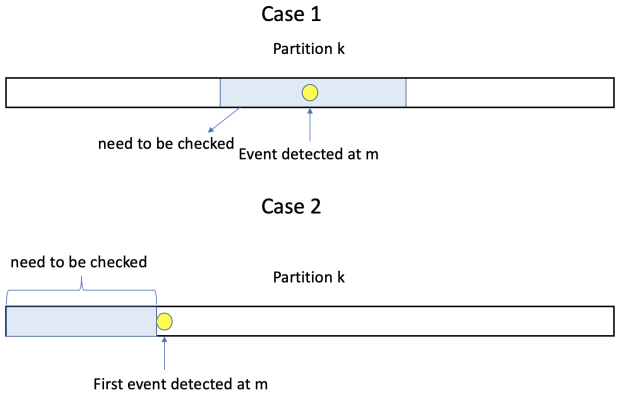

Figure 6: Verification cases

Let's first consider what causes these two problems. The answer is, when we run the event detection algorithm in parallel partitions,

the Window2 start position in each partition is different from when we run the event detection algorithm in a single thread. In the single-threaded event detection algorithm, the following two cases can affect the Window2 start position:

- When we detect an event, we clear the Window2 contents and the new Window2 start position is where the previous event is detected.
- When we do not detect an event after processing more than 100 features (Window2 size), the algorithm clears the Window2.

In a single-threaded program, both cases are fine since the Window2 start position can be updated from the previous state. However, in the data-parallel algorithm, each task cannot know where other tasks have possibly detected an event because they are executed concurrently on disjoint partitions of the input data (in fact they depend on the previous partition in a single-thread algorithm, although there is no partition concept here, we just use it to illustrate this issue). Without knowing the previously detected event information, each task starts its Window2 from where its partition starts, and this point can be different from the Window2 start position in the single-threaded algorithm.

We have concluded two cases where incorrect results might occur. These two cases are shown in Figure 6, the features around the detected event in Stage 3 output, and the features appear before the first detected event in each partition. To fix these incorrect results, we record and update the correct Window2 start position from the beginning of Stage 3 output results. When we encounter these two cases, we have a correct Window2 start position. Then we run the event detection algorithm again on the features included in these two cases and output the correct results.

Case 1 includes the features around the detected event in Stage 3 output. In the program, for example, if the detected event in Stage 3 has index m, case 1 will include features in the interval $[m-200, m+200]$. With the correct Window2 start position, we run the event detection algorithm on this interval and the output results are correct.

Case 2 includes the features which are the interval starts from partition head and ends at the first detected event in each partition. For example, the first detected event in partition k is at m, case 2 includes features in $[partition\ k\ start\ index, m]$. With the correct Window2 start position, we run the event detection algorithm on this interval and output the correct results.

The verification part only needs to run the event detection algorithm on the features included in these 2 cases, which is just a small fraction in the whole stream since events are sparse. Though this is processed in a single thread, it will not block the stream because the computation load is low. However, it might result in higher latency, as some input results from Stage 3 cannot be output immediately because we need to wait for all the features in each case to arrive in order to run the event detection algorithm.

4.5 Time analysis

Assuming the system is processing a dataset with N batches of the same size. It takes t_{data} time to request one batch data and convert it to a corresponding feature. The cost of processing event detect tasks on one feature is t. As event detect algorithm is applied on features during processing, t is much larger than t_{data}

In the sequential model of processing is shown in Figure 7, the overall run time of N batches of data is $N \times (t + t_{data})$.

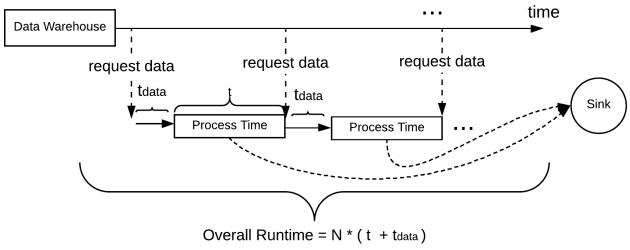

Figure 7: Time consumption of sequential processing

In our parallel implementation, suppose the system is working with a parallelism of P. Since the data batches are generated from a single-thread data source, and as we mentioned above, the time for generating one feature is t_{data}. therefore, each task thread needs to wait for time t_{data} to get features from the data source. Then for each parallel task that processes P batches, the average time cost of merging and verifying stage is t_{mv}. Since the distribution of events in data are sparse, t_{mv} is much less than the processing time t over one feature. As is shown in Figure 8, for every round of parallel processing over P features, the time cost $t_{mp} = t + (P-1) \times t_{data}$. So it spends $\frac{N}{P} \times t_{mp}$ processing the whole dataset of N batches.

Finally, we can derive an inequality (1) by an assumption: $t > t_{mv}\ and\ t > t_{data}$, and do the following steps:

$$
\begin{aligned}
&N(t + t_{data}) > \frac{N}{P}(t + t_{data}(P-1) + t_{mv}) \\
\Rightarrow &P \cdot t + P \cdot t_{data} > t + P \cdot t_{data} - t_{data} + t_{mv} \\
\Rightarrow &t(P-1) > t_{mv} - t_{data} \\
\Rightarrow &t > \frac{t_{mv} - t_{data}}{P-1}\ , where\ P \geq 2, P \in \mathbb{N}^{+} \qquad (1)
\end{aligned}
$$

- If $t_{mv} < t_{data}$, then the inequality (1) holds.
- If $t_{mv} \geq t_{data}$, since $t > t_{mv}$ and $t > t_{data}$ as we assumed, even if $P = 2$, the inequality (1) holds.

Therefore, we can conclude that the inequality (1) must hold under the assumption we made and our parallel strategy can always be faster than the naive strategy.

5 CHALLENGES

We faced many challenges while designing our strategy and implementing the parallel algorithm. One of them is transforming the event detector workflow into Flink stream processing operators in order to accelerate this algorithm by running it in parallel. Another challenge was tuning the trade-off between our verification part which is slow but achieves the best accuracy. In addition, the single-threaded implementation of the grader placed limits on the performance of our parallel algorithm. Finally, the missing data in Query 2 can hurt the achieved accuracy a lot. At first, we wanted to fit the function of the data since it is periodic, however, we found it to be extremely difficult. In this section, we clarify why these

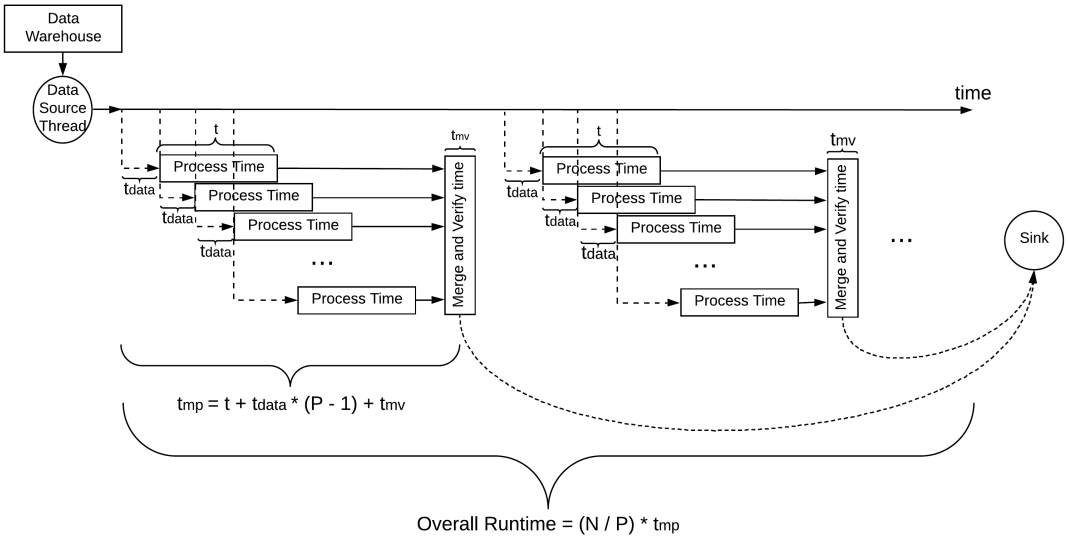

Figure 8: Time consumption of parallel processing

challenges occurred and provide some solutions or the decisions we made to overcome some of them.

5.1 Transforming the event detector into stream processing

In the beginning, we wanted to find a way to transform the event detector into Flink stream processing operators since this is one of the most time-consuming parts of our architecture. If we could break its logic into separate operators, we could boost its performance by leveraging Flink's task and operator parallelism. Unfortunately, we found this task to be very challenging to complete in the limited time we had in hand. Figure 9 shows how the workflow of the event detector is highly coupled, which makes it very difficult to transform this event detector algorithm into separate operators. Hence, we decided to keep its implementation inside a single operator and focus our optimization efforts on other parts.

5.2 Verification efficiency

Since the verification stage performs the major job of correcting the results from upstream, it needs to wait for a few events. Hence, the verification process may not be very efficient. However, our results show that our method achieves a reasonable speed and the highest accuracy. From our perspective, some of the performance overhead may be caused by the JVM which we could have avoided by fine-tuning. Besides, try implementing this verification part in C++ can also give possibilities of improving the speed performance. Therefore, our parallel algorithm and verification process can achieve a balance between high accuracy and speed in the meantime.

5.3 Single-thread grader

It is not efficient if we perform the synchronized HTTP post in the same thread which also processed data. Hence we send all the post requests into a post queue once we finished the data processing, then we started an individual thread to post our results if the post

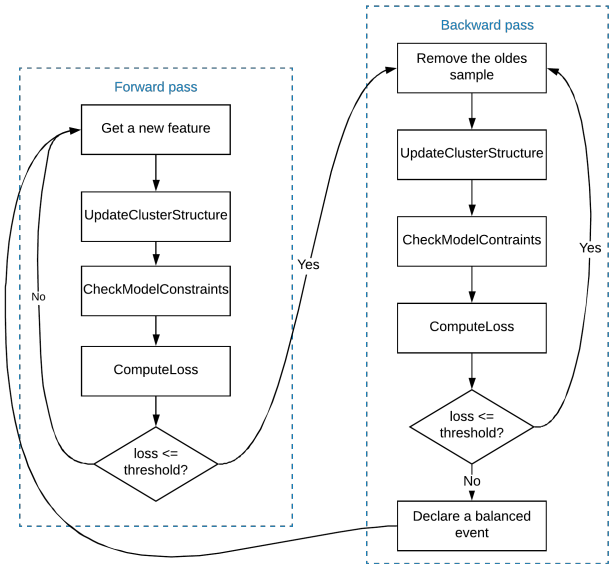

Figure 9: Flowchart of prediction

queue is not empty. This can improve the latency and total run time since we did not need to wait for the synchronized post to be finished and can start processing the next feature immediately.

Besides, since the data source provided by the grader is single-thread, the power of our parallel method can be maximized if we could perform concurrent HTTP requests.

5.4 Dealing with missing data

Since the data of Query 2 can be missing, we need to deal with this case so that the prediction can be more accurate. To handle

lost records, we initially sought an algorithm to simulate the lost data. As we can see in Figure 1, the voltage curve is almost the *sine* function since it is alternating (which is periodic), hence we could fit the voltage functions very easily. However, as we can see in Figure 2, it is very difficult to simulate current data because the current has a lot of randomnesses even if it is periodic. Therefore, we decided to simply copy the latest good feature, and compensate for the missing data from that good feature. We believe this is a good strategy since the lost data feature is singular so that it could be safe to deal with it through this approach.

6 CONCLUSION AND FUTURE WORK

We proposed a novel parallel method for this challenge and achieved both accuracy and speed by our unique verification method. Although we did not achieve the best speed, we do believe it is not the defect of our method and we can definitely improve the speed in the future.

There are a lot of tricks that we can do to optimize our work. The first thing as we mentioned above is trying to implement our method in C++ or Rust which are both high-performance languages or tune JVM performance as well as introduce AOT or JIT. The other thing that we can do is trying to modify our watermark strategy to heuristic watermark which can have a dynamic bond depends on the quality of the stream.

ACKNOWLEDGMENTS

To our mentor Vasiliki Kalavri and the committee who provided much help in this challenge. To Apache Flink developers and open-source community which provides a convenient data streaming processing framework.

REFERENCES

[1] Karim Said Barsim and Bin Yang. 2016. Sequential Clustering-Based Event Detection for Non-Intrusive Load Monitoring. *Computer Science & Information Technology* 6. https://doi.org/10.5121/csit.2016.60108

[2] Carbone, Paris, et al. 2015. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[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

[4] G. W. Hart. 1992. Nonintrusive appliance load monitoring. *Proc. IEEE* 80, 12 (1992), 1870–1891.

[5] F. Hueske and V. Kalavri. 2019. *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*. O'Reilly Media, Incorporated. https://books.google.com/books?id=64GHAQAACAAJ

[6] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3, Article 19 (July 2017), 21 pages. https://doi.org/10.1145/3068335