Grand Challenge: Using Streaming Data and Apache Flink to Infer Energy Consumption

Manuja DeSilva*
mdesilva@bu.edu
Boston University
Boston, Massachusetts

Michael Hendrick
mhendric@bu.edu
Boston University
Boston, Massachusetts

ABSTRACT

This paper entails the technical details of an approach to the challenge presented by the DEBS 2020 committee [5], regarding Non-Intrusive Load Monitoring (NILM) and its relevance in the area of data streaming. Our project highlights how the open source project Apache Flink can provide an efficient solution for processing large data-sets. Furthermore, we implement a version of DBSCAN, a data clustering algorithm, and we present an effective approach for handling out of order events in a data stream. We observe that our approach strikes a balance between optimization, usability, and accuracy with room for future work. We propose a complete solution that is capable of detecting appliance power events and energy consumption by using a stream of voltage and current data.

CCS CONCEPTS

• **Computing Methodologies** → **Modeling and Simulation**; • **Applied Computing** → *Physical Science and Engineering*; • **Mathematics of Computing** → Probability and Statistics; • **Computer Systems Organizations** → Dependable and Fault Tolerant Systems and Networks.

KEYWORDS

Data Stream Processing, DBSCAN, Event Modelling, Event Based Systems, Real Time Processing

ACM Reference Format:
Manuja DeSilva and Michael Hendrick. 2020. Grand Challenge: Using Streaming Data and Apache Flink to Infer Energy Consumption . 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, 4 pages. https://doi.org/10.1145/3401025.3401759

1 INTRODUCTION

Non-Intrusive Load Monitoring (NILM) [2] is the process of observing changes in voltages and currents in a building and using this information to identify what appliances are present in the building and how much energy those devices could be using. The 2020 DEBS Challenge[5] is focused on taking a stream of voltages and currents and identifying possible events using statistical modelling, as outlined in the reference paper [2]. The provided dataset gives information regarding voltage ($V = I \cdot R$) and current ($I = \frac{V}{R}$) values, as well as an unique *id* for each *event*. Our approach involves using Apache Flink [3], an open source platform that is designed from the ground up to offer low latency and high throughput in regards to large data streams. Table 1 highlights the significantly lower latency our solution has while only having a marginal increase in runtime. To achieve this, we take advantage of Flink's operators, discussed further in Section 3, alongside a DBSCAN detection algorithm [4] to compute results in a timely fashion. Figure 1 highlights an overview of our approach, and more detail is provided in Section 3.

*Both authors contributed equally to this research.

Table 1: Query 1 Comparison on Input of 500000 Events

Test	**Solution**	**Baseline**
Runtime	9243ms	68928ms
Latency	54ms	129ms
Accuracy	100%	100%

2 BACKGROUND

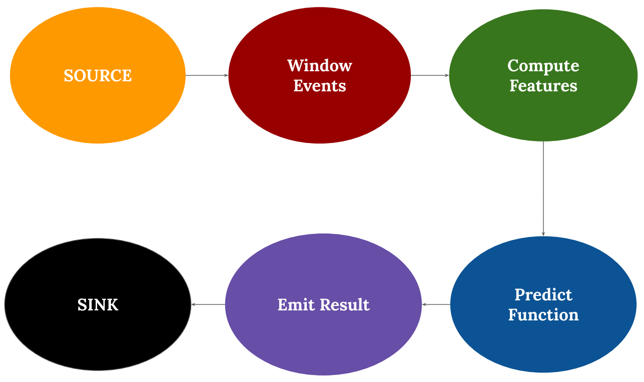

Figure 1: A broad look at our approach to the challenge

2.1 Apache Flink

Apache Flink plays a critical part in the architecture of our solution. At its heart, it is a lightweight platform designed to process batches of streaming data from various inputs, from files on disk to real-time sockets. Flink can also be adapted to take input from other external sources as well; our solution utilizes an Apache HTTP client to feed records to Flink. Many elements that one needs to consider when working with event streams, such as data distribution, fault tolerance, and managing time, all work out of the box with Flink. In particular, its windowing API proves to be very useful in that it automatically chunks all the samples into a window size of our choosing. Internally, this allows for each batch of samples to be processed very quickly, rather than if we had chosen to implement our own logic that would block until one batch had finished processing before starting the processing of the next batch. This configuration required little intervention on our part aside from defining the type of window and its size, permitting us to focus more on the core component of our program: the clustering algorithm.

2.2 DBSCAN

Density Based Spatial Clustering of Applications with Noise (DBSCAN) is a popular data clustering algorithm that when provided two arguments, minimum amount of points and an epsilon value, returns clusters, each of which contains sets of our input data deemed to be similar to each other; these are also referred to as *neighborhoods*. Samples that do not belong to any cluster are assigned to a *noise* cluster. We draw several metrics from each of these clusters, including its maximum member, its minimum member, and its temporal locality.

2.3 NILM

DBSCAN, as referred above, can be used as part of a Non-Intrusive Load Monitoring (NILM) solution. As detailed in the referenced paper [2], Sequential Clustering-Based Event Detection For Non-Intrusive Load Monitoring, the authors use DBSCAN to identify clusters within a growing window of features, even when the beginning and end of the window are not consistent or easily identifiable from the raw data itself. At a high level, in addition to the DBSCAN parameters, our adaptation of the NILM solution takes as input a temporal locality epsilon value and a loss threshold lambda value, and utilizes these parameters to find clusters that meet the event model constraints and ultimately pinpoint a non-interleaving cluster pair with minimum satisfactory loss.

3 SOLUTION ARCHITECTURE

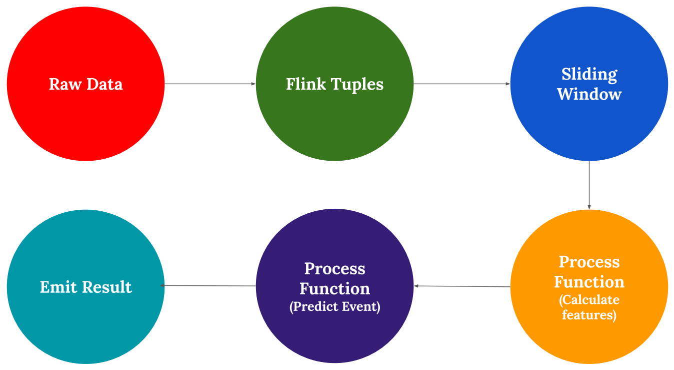
Raw Data
Flink Tuples
Sliding Window
Process Function (Calculate features)
Process Function (Predict Event)
Emit Result

Figure 2: An in-depth look in Apache Flink's involvement in our solution

We entail below the steps and components our solution takes to form and report results based on the events inputted into the stream.

3.1 Tuple Elements

Flink features its own implementation of a Tuple object out of the box, which comes optimized for Flink workflows and other operators. It is this Tuple element we convert our raw data into, and we use this structure throughout our body of work.

3.2 Flink Operators

Apache Flink is composed of many operators that transform data streams into new data streams, and combining operators allows developers to create intricate computing logic. We highlight the operators we use below, and in Figure 2.

3.2.1 Sliding Window. Sliding windows are Flink operators that group elements inside a window of fixed length. We use this in our solution to batch together elements a thousand at a time to detect possible events. Although we use tuple-based windows to batch our events, we use watermarks to facilitate the batching, as Flink uses the watermarks internally to determine the order of events and efficiently batch the incoming data. This helps streamline our workflow greatly, and allows us to separate event groups easily.

3.2.2 Process Function. Process functions in Flink do the necessary low level computational work to generate meaningful results. We use a process function inside our sliding window to calculate reactive and active power, and to generate the tuples that our predict logic is expecting. In addition, we use a separate process function to call our predict method and emit the solution for each window to the correct output.

3.3 Predicting Events

3.3.1 Definitions. Allow us to provide context on terms that appear in the reference paper [2] below.

- `temporal locality`: A ratio that explains how a given cluster is spreading over the time domain.
- `Model 1`: An event without a noise cluster and two non interleaving clusters.
- `Model 2`: An event that potentially has a noise cluster and two clusters with a high temporal locality ratio, both of which do not interleave in the time domain.
- `Model 3`: An event that potentially has a noise cluster and at least two clusters with high temporal locality, both of which do not interleave in the time domain.

3.3.2 Upstream operators. We begin by continuously retrieving record tuples from the source via sequential `HTTP GET` requests, and convert the raw input tuples into compact and efficient Flink *Tuple* objects that are optimized for quick serialization and deserialization operations throughout the life-cycle of our data stream. During the conversion, we also generate *watermarks* [1], a progress tracking feature of Flink that provides the stream a sense of time, permitting us to create window-based events. Window-based events allow us to easily define window *lengths* of an arbitrary sample size and then create features for each of these windows. For our experiment, we create a sliding window of 1000 events. For every 1000 events, we generate a tuple with the current window id, the calculated active power and the calculated reactive power features. Each of these tuples is then forwarded to our Event Detector class, which continuously processes a growing window of active and reactive features, running the Forward Pass workflow, and, if an event is detected, the Backwards Pass workflow.

3.3.3 Forward Pass. Our implementation of event prediction is one that closely models the approach described in the reference paper [2]. All prediction attempts begin by receiving a list of N consecutive events that are provided by the upstream operators described above. We then run our DBSCAN algorithm on this set of events, and return a set of clusters, each containing some unique set of events. Next, for each iteration of predicted clusters, in addition to verifying that we have at least two non-noise clusters, we verify that we have at least two clusters with a locality of ($locality > 1 - \texttt{temporal locality}$). Then, we look for a set of cluster pairs which share non-interleaving points, if such cluster pairs exist. It also returns an interval of noise events located between the two clusters. Using this data, the cluster pair and its event interval is checked against the acceptable model loss value. If it is validated, then an event is detected. If an event is not detected, then the forward pass begins again with $N + 1$ consecutive events. As a caveat, to avoid our growing window of events from growing arbitrarily large, we clear the window every 100 windows if an event has not been detected in the last 100 windows.

3.3.4 Backwards Pass. If the model loss was satisfactory, we declare that an event has occurred, and we attempt to determine exactly where the event transpired. This is done by removing the first sample from the list repeatedly while the loss is still satisfactory. Once we reach a stage where the loss is too high, or an event is no longer detected, we reinsert the removed sample and declare this range the location of the event.

3.4 Handling Out of Order and Late Arrivals

For the second part of the challenge [5], we were required to devise a solution that could account for some events arriving late, or possibly not even at all. Originally, we approached this by waiting 20,000 time units to see if the events would eventually arrive, and if so calculate per normal otherwise progress. As one might expect, this led to very high accuracy despite the late arrivals, but however also led to higher latency, in part due to the waiting, and in part to how Flink handles the batching of data. In our final solution, we decide not to wait for those tuples, and instead we fill in missing data with our own data. The data that we populate the incomplete windows is based on the average of what has been seen up until things went missing, multiplied with some noise to create pseudo random data points. When testing, we found that while this was not as accurate as waiting for incomplete windows, it was still detecting most events correctly, with a profound increase of timeliness. This appeared to us as a reasonable trade off, and is one we moved forward with.

4 DOCKER INTEGRATION

When building our solution into the requested Docker container format, we first approached the build by following the official Flink container guidelines [1], which posted that we run the job manager and task manager on separate containers, and our application in an additional container. However, due to the restrictions on the docker-compose format, we packed the job manager, task managers, and application into one container. When the container is instantiated it triggers the Flink cluster, which encompasses both the job manager and task managers, to start. After determining the grader infrastructure is ready, our application submits its "job" to the cluster and runs our prediction algorithm on the incoming tuples.

5 CHALLENGES FACED

In particular, we found our shortcomings in two specific areas. The first of which was implementing NILM as described in the reference paper [2]. Whenever implementing an idea presented in another work, great care must be taken in ensuring that the created version mimics the original to a high precision level. Unfortunately, there was a minor part of our version that was off from the original, that led to slight but noticeable errors causing a dip in the accuracy of our implementation, even when events were in order, most often when we scaled to larger datasets. This decrease in accuracy is comprised of detected events not listed in the ground truth. So, our algorithm and its underlying model may have been over-fitted in some aspects. In addition, Apache Flink comes with a lot of great, useful features out of the box that helped immensely when devising our solution. At the same time, we were constrained to the limits of the system, in particular the batching of data. This batching process we believe led to an increased latency, and is an attribute of our solution that we found we could only improve up to a certain limit.

6 CONCLUSION

Our solution obtained results with a high degree of accuracy and low latency in comparison to the baseline solution. By stitching together existing software and testing ideas with our own unique additions, we have put together a platform that can easily be built upon and improved. While our final results and model may have over-predicted events in some cases, we maintain faith that this software can be used as a building block for future work in this space.

7 FUTURE WORK

While we are pleased with the timeliness of our solution, we would want to spend more time ensuring the accuracy of our solution, particularly at large scales. As our solution stands, we trade misinformation for lower latency, and we believe that there exists approaches where the trade off does not need to be as stark as it currently is in our implemented solution. We would also want to explore a scenario of multi-threading and avoiding cases of repeat work. Most likely, our current DBSCAN is a bottleneck in the overall performance of the system, and performing some optimizations here would lead to a system wide increase in performance.

[1]https://ci.apache.org/projects/flink/flink-docs-stable/ops/deployment/docker.html

8 ACKNOWLEDGEMENTS

We would like to thank Professor Vasiliki Kalavri (vkalavri@bu.edu) for introducing us to the world of data streams and processing. We would have never achieved what we did without her guidance, and we are very appreciative for the lessons she taught us along the way.

REFERENCES

[1] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proceedings of the VLDB Endowment* 8 (2015), 1792–1803.

[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] 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 Eng. Bull.* 38 (2015), 28–38.

[4] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (Portland, Oregon) *(KDD'96)*. AAAI Press, 226–231.

[5] 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