Grand Challenge: Anomaly Detection for NILM Task with Apache Flink

Zongshun Zhang*
zhangzs@bu.com
Boston University

Ethan Timoteo Go*
ethango@bu.edu
Boston University

ABSTRACT

The topic of the 2020 DEBS Grand Challenge is to develop a solution for Non Intrusive Load Monitoring (NILM). Sensors continuously send voltage and current data into a stream processing application that would detect the pattern of power data based on the data characteristics. NILM is important in signal processing especially in those advancing areas such as 5G and IoT products, which generate massive amounts of data from the edge of the network. Our solution focuses on how to divide and parallelize jobs as small as possible while keeping some reasonable Service Level Agreement (SLA) including job sizes and latency so that it would be practical for edge or fog deployment. This paper describes our solution based on Apache Flink, a stream processing framework, and the DBSCAN density based clustering algorithm for anomaly detection through the context of data provided by DEBS Grand Challenge.

CCS CONCEPTS

• **Networks** → Cloud computing; • **Computer systems organization** → Real-time system architecture.

KEYWORDS

cloud, computer systems, real time systems, NILM, DBSCAN, Apache Flink, edge computing

ACM Reference Format:
Zongshun Zhang and Ethan Timoteo Go. 2020. Grand Challenge: Anomaly Detection for NILM Task with Apache Flink. 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, 5 pages. https://doi.org/10.1145/3401025.3401758

1 INTRODUCTION

Non-Intrusive Load Monitoring (NILM) [6] is a signal processing application to collect and analyze load data in a non-intrusive manner. The NILM method for voltage and current discussed in this paper can be generalized to many applications, such as cloud or network system load balancing. The same idea can be applied to measure variance in the utilization of data centers and their corresponding inter and inner DC communication. For example, routes need to handle transient loads depending on the time of the day or district-specific hot-spots. NILM can help to design dynamic scheduling and deployment strategies to reduce cost and mitigate Service Level Agreement (SLA) violations, for example: by accessing close computation resources or scaling down resources.

This year's DEBS challenge[11] aims at implementing a NILM system to detect abrupt changes in electric power. Solutions are ranked based on total run time, accuracy, latency, and timeliness. A proper implementation needs to strike a balance between two tradeoffs: (i) total run time and latency for Query-1, and (ii) timeliness versus accuracy for Query-2.

The environment supplies electric current and voltage data in batches, including out-of-order records, and the solution needs to detect *power-value-pair-change*. In this paper, we examine the idea of dividing jobs into simple short tasks, potentially deployed on edge or volunteering nodes. With the ubiquity of 5G and IoT devices, we expect underutilized computation capacity to increase in the society[10]. For example, there will be a much higher density of 5G base stations across the globe, which can be used temporarily as edge computing resources for simple pre-processing jobs.

Our solution consists of three steps. First, we accumulate out-of-order records. Second, we merge and order all the records for the event detection process. Third, we use the DBSCAN clustering algorithm to detect *power-value-pair-change* events a specific window size.

Our solution is based on Apache Flink. It is an event-driven stream processing system on the top of JVM. We use Apache Flink [8] to setup the computation as a directed acyclic graph (DAG). On the graph, we can specify transformations to be applied to the input stream in a specific order. Also, we leverage Flink's operator and task parallelism to divide the queries into smaller tasks that can be executed concurrently. The incoming request stream is distributed to different tasks of each operator, and until the data flow reaches the *power-value-pair-change* event detection operator, the results are pushed to the grader. DBSCAN clustering algorithm is performed to infer the detection results.

Our Query-1 solution assumes that there are not out of order events. We apply our solution through the operators without any out-of-orders policy. On the other hand, in Query-2, we assume that there are out of order events and implemented a decision threshold to keep or drop events.

2 BACKGROUND

The task is to process a signal stream, an unbounded sequence of voltage and current records that are generated by a remote data source. The input stream needs to be transformed to extract a set of

*Both authors contributed equally to this research.

DEBS '20, July 13–17, 2020, Virtual Event, QC, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8028-7/20/07…$15.00
https://doi.org/10.1145/3401025.3401758

features and then clustered to detect patterns that might indicate changes in power.

With paper [6] and baseline solution provided to us, the proposed an algorithm that uses forward and backward propagation steps with some transformations to identify *power-value-pair-change* events.

2.1 Event Model

The event detection pattern is motivated by a previous work discussing NILM with the clustering method [6] and the baseline solution[1]. There are three models mentioned in that paper. The first model matches a case when there are two distinct clusters of power data in the data stream seen so far. The second model considers one step forward that there can be noise in each of the two data clusters while the *temporal locality*[6] of each cluster is high; i.e., close to 1.0 which means there are not many noisy power pairs in the time range of each cluster, so each of the clusters is believed to be a *stationary segments*[6]. The third one is a further generalization that there can be at least two high *temporal locality* clusters with little noise. Our solution is based on this third model, which we describe in detail below.

Assuming there are at least two high *temporal locality* clusters in the data stream have seen so far, there can be more than one *power-value-pair-change* events. In this case, we would need to go through the data in the event time order and return the event detection result for all two consecutive clusters, if they exist. We name those steps *event model checking procedure* in this paper. There are three constraints in this model that were checked for our *power-value-pair-change* event detection. First, we want to make sure at least two clusters are present. Second, we would need to make sure the *temporal locality* of each cluster to be close to 1.0 so that fewer noises are in the time range of each cluster. Third, to avoid clusters with different power values but at the same time domain, we would like to make sure the clusters have a distinct state change interval such that the latest data point of one cluster is earlier than all data points of the other cluster. When all those three conditions are satisfied, we claim a *power-value-pair-change* event is detected. Our implementation takes the baseline solution and furthermore implement Flink's multi-threading capability. In order to do so, we translated the code to fit Flink's multi-threading structure.

2.2 Apache Flink

We take advantage of the Flink stream processing framework so that we do not need to worry about all the low-level implementations and data flow, and we can focus on the algorithm.

Flink's programming model is a directed acyclic graph(DAG) [7] and has two levels of abstraction. The upper-level abstraction is called a job graph[4]. This is where a user can define the DAG and the functionality of each operator inside the graph. The lower level abstraction is called the execution graph[4]. This is the actual data flow being executed by the system. Flink will convert a user's job graph to an execution graph based on the execution environment set up by users. This is similar to converting a DB query plan to an execution plan.

Flink's event-driven nature helps us keep a balance between latency and parallelism by operators. This framework provides the necessary functionality required by our solution, including operator and task-level parallelism, watermarks, event-time timestamps for handling out-of-order records, and key-based partitioning. We discuss those tools and how we use them next.

2.2.1 Operator and Task Parallelism. Users can specify a parallelism number for each operator in the job graph. Flink would then create the corresponding number of tasks, each having its thread. We take advantage of this feature to save time on implementing low-level threading details.

2.2.2 Watermarks and Timestamps. Flink has built-in event time management. It supports the notions of Watermarks and Timestamps. A Timestamp is a property attached to every element transmitted in the dataflow. It indicates the time of the element in the event time domain. Watermarks are unique records that provide progress metrics of the dataflow to operators. Each operator has a dedicated method to handle watermarks and decide whether it has received all required input and whether it is safe to output results to downstream. This is an important feature that we can use to carry out event time-domain transformations.

2.2.3 JobManager and TaskManager. Apache Flink designed two separate agents to handle different functionalities. This is a master and worker model. One is called JobManager, and the other one is called TaskManager. JobManager is the master, and it would need to deploy jobs and send health checks to TaskManagers. Each time a job arrives, JobManager would convert the job's DAGs to execution graphs and schedule the job on each TaskManager to carry out the computations. Also, it has all the fault-tolerant features, but those are not expected to be used in this evaluation setting. TaskManagers are workers. Each of them has some slots for each task. The notion of a "slot" defines the number of threads that can operate concurrently in one TaskManager. This is a distributed architecture and theoretically it can save our time on handling machine communications. While the evaluation platform does not support this feature, we try to investigate the benefit of it.

2.2.4 Data Partitioning. Flink's built-in group by key mechanism, the keyBy() method, is helpful to scale and migrate data. After applying the group by key transformation, data would be hashed to each task of the next downstream operator based on a key specified by the user. Moreover, each parallel instance can manage several key groups at the same time. Each invocation of the method on the task is bounded to a key group. This makes the merging step very simple in terms of code but inefficient in terms of the data flow design. This group by hashing and reorganize causes overhead to the data flow.

2.3 DBSCAN Clustering

DBSCAN[9] is a popular density-based unsupervised clustering algorithm. Given a distance metric, the algorithm would try to calculate the distance between data entries and would try to put the data entries with close distance in a single cluster. A cluster is a set holding those data entries. The definition of closeness is also a parameter that can be configured. It is called ϵ neighborhood. If any 2 data entries are close enough or say in the "ϵ neighborhood" of each other, they would be put in a single cluster. However, to

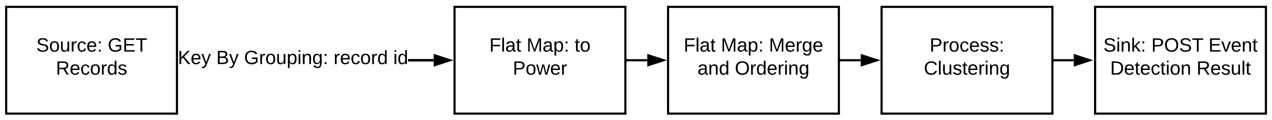
Source: GET Records
Key By Grouping: record id
Flat Map: to Power
Flat Map: Merge and Ordering
Process: Clustering
Sink: POST Event Detection Result

Figure 1: Query-1 DataFlow

remove noisy data, DBSCAN also constrains the number of data to form a cluster. This is called "minPts," which defines the minimal number of data entries that can form a cluster.

We are using the Apache Math3[2] implementation of the DBSCAN algorithm for abrupt *power-value-pair-change* event detection. We are matching the third model mentioned in section 2.1. We begin with packing each corresponding active and reactive power value as a point. Then, we cluster on the points with a pre-defined euclidean distance and number of minimal neighbors parameters. Next, we find some significant different clusters among those clusters that indicate an abrupt change in power values by checking the constrains mentioned in section 2.1. Then we output the *power-value-pair-change* event detection result.

This implementation of the DBSCAN algorithm would compute a new clustering decision from scratch every time we input the power points. This is a place that we spent much time to study a job dividing strategy to alleviate this bottleneck. We were trying in 2 directions. First, we tried to reuse previous clusters to save time comparing clusters from scratch each time we get a new PowerPoint. Second, we were discussing, after the merge and ordering phase, we could fetch a clustering job each time when we got a new PowerPoint to save time with parallelism. The second approach would create a lot of duplicate works. We discuss this further in section 4.

2.4 Docker

The evaluation platform is based on Docker[12]. The graders provided a docker-compose template for us to pull our solution image and their grader image as two services which is actually two containers. They can communicate with the common HTTP connections over TCP.

Docker is a virtualization method that can create a virtualized environment to run an isolated environment with libraries and codes. It also utilize local resources like OS kernel and some other low level libraries to handle IO and other operations that can be shared among containers and supported by the local OS.

Docker Compose is a tool to deploy multiple containers at the same time[5]. Users can specify how to run the services, which are actually containers, including the docker image, ports to expose, network, etc.

3 SOLUTION ARCHITECTURE

Overall, the 2 queries have many similarity in the transformations. First, each record is partitioned with its id, which is the event time timestamp generated by the grader. Afterwards with each id, there is an accumulator for all the records related to this partition. The accumulator has a hard-coded latency bound, and it would output the records in a batch when its watermark is passing the bound. Any record that came to the accumulator later than this bound would be dropped. Then each batch is transformed and aggregated from voltage and current values to be active and reactive power values. Furthermore, there is a single ordering operator to order each power value set by their timestamps. After that, we use a DBSCAN implementation from Apache Math3 to cluster power values to find an *power-value-pair-change* point.

3.1 Query-1 Dataflow Construction

Figure 1 shows the dataflow for our implementation of Query-1. There are five operators in this DAG. one Source to get a batch of input data. Next, there is one Map operator to pre-process voltage and current to active and reactive power with high parallelism. Following there is one Map operator that merges and orders power data pairs. Then, a Process function applies clustering on the ordered pairs and finally, a Sink operator posts the result back to the grader.

Each of the operators has a parallelism parameter, which specifies the degree of parallelism of this operator. For the Source and Sink, to avoid duplicate data, we set parallelism to be one. The Process Function is also configured with parallelism one, since our DBSCAN clustering implementation is sequential. A higher parallelism would only duplicate the clustering computation. The remaining available threads can be used for the first Map operator, because it is simply transforms each batch to a pair of a power data points. Then, we have a lightweight merge and ordering operator to order power pairs for clustering.

The DBSCAN operator receives pairs of power points, and triggers a new *event model checking procedure* for each input. It then checks the 3 constrains for a *power-value-pair-change* event. Each of those computations is sequential, so a clustering can be triggered only after the previous one is finished. On each invocation of the DBSCAN clustering, the operator would try to cluster all data points it has seen for far. The data structure saving all the data, which is called Window 2 in the problem statement[11], would be cleaned up for memory management in an invocation satisfying two requirements, i.e. there is one *power-value-pair-change* event detected, and the size of this data structure exceeds a pre-defined boundary. Finally, the clustering operator outputs an event detection result to the Sink operator downstream, no matter if an event is detected or not. As the Sink operator receives a result from the clustering operator, it sends each result with a POST request to the grader at the corresponding HTTP address for Query-1. At the end

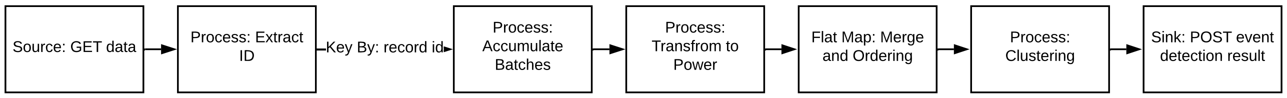
Source: GET data
Process: Extract ID
Key By: record id
Process: Accumulate Batches
Process: Transfrom to Power
Flat Map: Merge and Ordering
Process: Clustering
Sink: POST event detection result

Figure 2: Query-2 DataFlow

of the process, the sink operator sends a GET request to the grader system to trigger evaluation.

3.2 Query-2 Dataflow Construction

Figure 2 shows the dataflow for our Query-2 implementation. The dataflow consists of seven operators with functionalities similar to those of Query-1. A single-thread Source retrieves batches of input data and sends them to a parallel Process function which assigns the id of each record to be its key. This stage is executed in parallel using the default number of parallelism i.e. all available threads [3]. Later another Process function accumulates all records with the same key, which is done by the keyBy(), the group-by key mechanism in Flink mentioned above. This operator is used to wait for out of order records, and we have a event time domain timer to set how long to wait and then emit the batch. The group-by key mechanism is used to sort records by their timestamps. Then the steps are very similar to what we have in Query-1. We have one more Process function to pre-process voltage and current to active and reactive power from each batch, deployed with the default parallelism, and then a Map operator merges and to ordered each power pair set. A Process function then applies the *event model checking procedure* and pushed the result to a Sink operator which posts the result back to the grader.

3.3 Evaluation Trigger

To correctly trigger the evaluation, we need to apply a special handling in the source data GET and result from POST methods. The benchmark environment is a flask HTTP server in Docker Container. The solution can send a GET request to get each batch of input data. To push each event detection result back to server, the solution needs to send a POST request with the result data. Depending on the path those REST requests are sent to, i.e. "/data/1/" and "/data/2/", the server would react with data for Query-1 and Query-2, respectively. If the end of the batch is reached, the server will send one additional empty string as an indication for the solution to stop retrieving the batch of data. Moreover, when the graders receive another GET request, the server will trigger its grading procedure. This took us some time to debug, as if it is triggered too early, which is possible in our case as we are utilizing operator level parallelism, the grading system would find our result was not complete. Our solution to that is to have our Source operator pass that empty string downstream. Furthermore, in the Sink operator, if that empty string is found, it can safely send a GET request to server to trigger the evaluation. In our solution, we implement the two functionalities in our source operator and sink operators. Source and Sink are the beginning and destination in Flink data flow, and we can specify the transformations the two operators can carry out. We have a loop in the source operator to keep reading batches of input data until an empty string is received. Furthermore, in the sink operator, whenever there is some result generated from upstream, it would send the result to the corresponding path for a different query with a POST request.

4 EVALUATION AND DISCUSSION

Our solution was ranked sixth with total rank twenty-one in the competition. In Query-1, our total run time is shorter than the baseline solution, while the latency is longer. This makes sense as in our multi-threading implementation, each operator's throughput is different, and the Source operator does not need to wait to get the next data set. While for Query-2, our timeliness rank was worse than the baseline solution, the accuracy is better. This is expected as we have an additional step to wait for out-of-order records to help event detection.

When we submit the final version, we set the solution of Query-2 to only take advantage of the operator parallelism, as the cost of multi-threading for this benchmark-setting is far higher than its benefit. Optimizing latency is very challenging in our setting. While we make more threads to partition the Job, the time for context switches, data deliveries, and waiting for bottleneck operator becomes significant. We discuss the tradeoffs later in this section.

4.1 Latency and Timeliness

The baseline solution works with a single thread. Each time a batch of records arrives, it saves it in its W2 window and triggers clustering for event detection. Moreover, if there are out-of-order records, it ignores them and pads 2.0 to those missing values. This design provides good latency and timeliness performance but bad total run time, as no CPU cycles are wasted when waiting for bottleneck operators. On the other side, when we try to partition jobs to small components, this can help to improve utilization of the benchmark system and achieve a shorter overall duration. But it would introduce extra latency in terms of each record. We believe our solution would work better in our hypothetical edge computation environment with some useful scheduling algorithms. However, this is not directly aiming at the evaluation criteria of this competition, and in some sense, we were off-topic when we decided on the design.

4.2 Parallel DBSCAN

There is another parallel DBSCAN algorithm; for example, in [13], they discussed a DBSCAN method with region growing and merging connected components. With task-level parallelism in Flink, we can have a clustering job initiator to receive new power data pairs. Moreover, when each new pair arrives it can initiate a new clustering task which is in an individual thread for *power-value-pair-change* event detection on the data stream have seen so far. Nevertheless, in the NILM setting, we always need a merge and ordering step to order power data pairs for clustering. The bottleneck caused by this step is more than the benefit given by parallelizing the DBSCAN algorithm. However, in the hypothetical setting, as the ordering step needs a node that has enough storage but little computation power, it can be deployed in a regional DC. Then it can distribute ordered points to local edge nodes to do DBSCAN clustering in parallel and find another node to merge those parallel clusters back for event detection. Then this can be the better option of the DBSCAN algorithm because it can help to utilize edge computing capacity and not only to use the capacity in a data center.

We spent much time studying how to reuse clusters, which prevented us from investigating more exciting ideas, such as parallelizing the DBSCAN algorithm. We thought about this parallel method several weeks before the deadline of our submission, but until the submission, whether the merge and ordering bottleneck would destroy all the benefits of this parallel gain remains unclear.

Comparing these two methods, they both save time for waiting for a *event model checking procedure* to finish. One enables reuse and avoids the re-calculation, and the other one would parallel the computations. However, we do not know which one would be better to implement in the real world.

4.3 Docker Compose

One constraint in the docker-compose template is that we can only set up one *service* for our solution. However, the recommended setup for Apache Flink is to have two *services* running separate functionalities, one is JobManager, and the other one is TaskManager. These constraints let us have a single JVM to run the functionalities of both *services*, which can lead to performance degradation.

4.4 The Policy to Update W2

The policy of updating W2 happened to be too constrained in Query-1. It limits the ability to parallelize jobs. The only case to clean up W2 is that no *power-value-pair-change* event is detected, and the window size exceeds a pre-defined parameter. So each clustering job would need to wait for the result of the previous clustering job. We do not see the difference between this algorithm and clean up when W2 reaches its pre-defined limit. Additionally, neither can detect the particular case that right after cleanup, a *power-value-pair-change* happens. This lets us match the baseline solution's event detection result rather than research how to get better performance.

5 CONCLUSION

Evaluating the performance of our query, we are elated to have experience developing a solution for this project. We studied the idea of utilizing edge computing nodes for the NILM system and the tradeoff of this idea. While we were unable to optimize our code fully with our strategy, we believe that the idea is viable to the problem given by the DEBS challenge, especially in a more general WAN environment. Apache Flink is just one of many tools that can be used to create solutions to the challenge. Our hope for the future is to combine many different tools and ideas in developing an efficient stream processing application.

REFERENCES

[1] 2020. *flink-default-parallelism.* Retrieved June 3, 2020 from https://github.com/dmpalyvos/debs-2020-challenge-local/blob/master/solution_app/Event_Detector.py

[2] 2020. *flink-default-parallelism.* Retrieved June 1, 2020 from http://commons.apache.org/proper/commons-math/javadocs/api-3.6.1/index.html

[3] 2020. *flink-default-parallelism.* Retrieved May 24, 2020 from https://github.com/apache/flink/blob/ab947386ed93b16019f36c50e9a3475dd6ad3c4a/flink-streaming-java/src/main/java/org/apache/flink/streaming/api/environment/StreamExecutionEnvironment.java#L252-L266

[4] 2020. *JobManager Data Structures.* Retrieved May 23, 2020 from https://ci.apache.org/projects/flink/flink-docs-release-1.10/internals/job_scheduling.html#jobmanager-data-structures

[5] 2020. *Overview of Docker Compose.* Retrieved May 1, 2020 from https://docs.docker.com/compose/

[6] 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

[7] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *Proc. VLDB Endow.* 10, 12 (Aug. 2017), 1718–1729. https://doi.org/10.14778/3137765.3137777

[8] 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.

[9] 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.

[10] Julien Gedeon, Michael Stein, Jeff Krisztinkovics, Patrick Felka, Katharina Keller, Christian Meurisch, Lin Wang, and Max Mühlhäuser. 2018. From Cell Towers to Smart Street Lamps: Placing Cloudlets on Existing Urban Infrastructures. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 187–202.

[11] 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

[12] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[13] Mostofa Ali Patwary, Diana Palsetia, Ankit Agrawal, Wei-keng Liao, Fredrik Manne, and Alok Choudhary. 2012. A New Scalable Parallel DBSCAN Algorithm Using the Disjoint-Set Data Structure. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Washington, DC, USA, Article 62, 11 pages.