Grand Challenge: Incremental Stream Query Analytics

Pedro Silva, Wang Yue, Tilmann Rabl
{firstname.lastname}@hpi.de
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany

ABSTRACT

Applications in the Internet of Things (IoT) create many data processing challenges because they have to deal with massive amounts of data and low latency constraints. The DEBS Grand Challenge 2020 specifies an IoT problem whose objective is to identify special type of events in a stream of electricity smart meters data.

In this work, we present the *Sequential Incremental DBSCAN-based Event Detection Algorithm* (SINBAD), a solution based on an incremental version of the clustering algorithm DBSCAN and scenario specific data processing optimizations. SINBAD manages to calculate solutions up to 7 times faster and up to 26% more accurate than the baseline provided by the DEBS Grand Challenge.

CCS CONCEPTS

• **Information systems** → **Data streaming**; *Clustering*.

KEYWORDS

DEBS Grand Challenge, stream processing, event detection, clustering

ACM Reference Format:
Pedro Silva, Wang Yue, Tilmann Rabl. 2020. Grand Challenge: Incremental Stream Query Analytics. 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/10.1145/3401025.3401756

1 INTRODUCTION

The proliferation of the *Internet of Things* (IoT) is changing the way people interact with devices, services and institutions. The amount of connected devices and the data they generate creates new challenges in terms of data processing: in 2020, it is expected that more than *30 billion* devices will be sending data over a network [1]. Many IoT applications, such as *Industry 4.0*, *smart home*, and *health care* applications have real-time constraints, adding another layer of complexity to those challenges.

The DEBS Grand Challenge 2020 [4] specifies an IoT use-case in which data produced by *electricity smart meters* are processed with the purpose of identifying special type of events. The quality of the challenge solutions is tied to the amount of correctly identified events and the time taken to identify them.

In this paper, we present the *Sequential Incremental DBSCAN-based Event Detection Algorithm* (SINBAD). It is based on the *Incremental DBSCAN* (iDS) [2], a re-implementation of the *the density-based spatial clustering of applications with noise* (DBSCAN), and two additional strategies, *Quick Result Calculation*, and *Noise Buffering* which speeds up the data processing on specific scenarios. SINBAD is up to 7 times faster and more than 26% more precise than the baselines we used, including an example implementation made available by the DEBS Grand Challenge 2020.

We state the problem and discuss background concepts in Section 2. We present SINBAD in Section 3 and its evaluation in Sections 4 and 5. We present related work on similar approaches from the state-of-the-art in Section 6 and conclude this work in Section 7.

2 THE EVENT DETECTION PROBLEM

The Grand Challenge [4] states the problem of inferring energy consumption profiles based on data available in a stream of smart meter measurements. The main challenge is to identify special events related to abrupt changes of consumption which characterize a profile change of the monitored appliances[1]. The Grand Challenge also introduces a solution, to which we refer as *Grand Challenge baseline* (GCB), and a straightforward Python implementation[2].

The GCB consists of (i) aggregating smart meter data in batches, (ii) generating features from each batch, (iii) clustering those features using DBSCAN and, (iv) analyzing those clusters for events.

In the next sub-sections we present more details about the GCB and the scenarios deployed for testing it.

2.1 Datasets and scenarios

A smart meter measurement is a tuple $< i, v, c >$ containing an id i, a voltage measurement v and a current measurement c. The identifier i is a sequential integer indicating the order in which a tuple was produced.

The Grand Challenge defines two scenarios, *Scenario 1* and *Scenario 2*, which define the characteristics of the stream of smart meter measurements. In Scenario 1, measurements in the stream are ordered by i and, in Scenario 2, data is incomplete (e.g., tuples with id 42 and 45 are present but not those with id 43 and 44) and disordered (e.g., tuple with id 57 appears before tuple with id 56). The Grand Challenge also defines that disordered tuples from Scenario 2 may be at most 20000 positions away from their ordered position. For example, the tuple with id 1 may be at most at position 20001 and tuple with id 42 may be at most at position 20042. This is

[1] https://spectrum.ieee.org/tech-talk/telecom/internet/popular-internet-of-things-forecast-of-50-billion-devices-by-2020-is-outdated

[2] https://github.com/dmpalyvos/debs-2020-challenge-local

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

ACM ISBN 978-1-4503-8028-7/20/07…$15.00
https://doi.org/10.1145/3401025.3401756

an important information that helps to differentiate missing from late tuples in online use cases.

Finally, there is one test dataset for scenario Scenario 1 containing 15M measurements, and another one for Scenario 2 containing 14915533 measurements, i.e., it has 84467 fewer measurements than the dataset of Scenario 1.

2.2 DBSCAN

In this section, we briefly describe the *density-based spatial clustering of applications with noise* (DBSCAN) [3], which is used in the Grand Challenge for clustering events.

DBSCAN groups together points which are in a neighborhood defined by a distance function and a given distance threshold. A point q_i is *reachable* from point q_j if q_i is within distance ϵ or smaller. A *path* $p(q_i, q_j) = q_i, p_1, \ldots, p_n, q_j$, between points q_i and q_j, is defined if each point in the path is reachable by its predecessor and successor. A set of points $q_1, q_2, ..., q_k$ in $\mathbb{R}^n$ belong to a cluster C if for every pair of points in C there is a path $p(q_i, q_j)$ of size greater than or equal to λ. Points which do not belong to any clusters are considered noise/outliers.

In the GCB, DBSCAN is used with the following parameters: (i) the minimum size of a cluster is 2, i.e., $\lambda = 2$; (ii) the distance function d is the euclidean distance; and (iii) outliers are stored in cluster C_0.

2.3 Grand Challenge Baseline

The Grand Challenge baseline (GCB) receives as input a stream of smart meter measurements and outputs sets of points, which characterize an event of interest.

The first step of the GCB is to aggregate measurements from the stream in batches of size 1000 and calculate one pair of *active* (P) and *reactive* (Q) powers per batch using the following functions (v_{RMS} and c_{RMS} are the root-mean-square values of voltage and current per period): $P = \frac{\sum(v \times c)}{1000}$ and $Q = \sqrt{(v_{RMS} \times c_{RMS})^2 - P^2}$.

Each tuple $< P, Q, id >$ containing one pair of active and reactive powers and an identifier is called a *feature*. The id of each feature indicates its temporal order associated to the batches which compose it. Those features are stored in a *window*, which we refer to as the *feature window*, and are processed by an algorithm called *sequential clustering-based event detection algorithm* (SECBED). It uses DBSCAN for generating clusters from the data available in the featuring window and analyzes those clusters for events.

2.3.1 SECBED. The SECBED has two main parts, the *forward search step* and the *backward search step*.

In the *forward search step*, whenever a new feature is available in the feature window, DBSCAN is applied to the entire data of the window in order to arrange the features into clusters. Those clusters are then used as input for validating an *event model constraint* – model M_3 available in [1] – which, when satisfied, result in an event detection. In short, that event model constraint guarantees that (i) there are at least two non empty non-noise clusters in the set of clusters; (ii) C_0, the set of noise clusters (cf. Section 2.2), is not empty; and (iii) the two first clusters C_1 and C_2 do not overlap in the time domain. This means that, if there is an event between clusters $C_1 = \{c_1^u, \ldots, c_1^v\}$ and $C_2 = \{c_2^n, \ldots, c_2^z\}$, where the superscripts indicate the temporal order of the features, and $u < v < n < z$, then no point $c_1^x \in C_1$ such that $x > v$ exists. Likewise, no point $c_2^y \in C_2$, such that $y < n$ exists. Finally, event features must be in the temporal interval between v and z. We refer to that interval as the *interval of the event*. Once an event is detected, the algorithm moves to the backward search step. Otherwise, it waits for the next feature and restarts the forward step.

In the *backward search step*, the objective is to find the smallest set of features that still characterize an event. To achieve this while the model constraint is still satisfied, it keeps removing the oldest feature added to the window and uses DBSCAN on the resulting window. The result of the algorithm is the set of noise features located between the two first clusters. It is expected that each feature has a result associated with it: a tuple $< s, d, e >$, where s is the id of the feature, d a boolean indicating if the feature is part of an event, and e, the median of the ids of the features describing an event, if any.

3 SOLUTION: SINBAD

The *sequential incremental DBSCAN based event detection* (SINBAD) is the algorithm proposed to solve the Grand Challenge.

3.1 Overview

SINBAD also batches smart measurements from the input and transform them into features $< P, Q, id >$, which feed the forward and backward search steps. Similarly to the GCB, both steps employ a clustering algorithm for processing a feature window, however, SINBAD replaces DBSCAN by the *incremental DBSCAN* (iDS) [2].

In the forward search step, iDS is executed whenever a feature is added to the feature window. When an event is found, the backward search step starts and the oldest features are removed from the feature window, one by one, until the smallest set of features that still characterize an event is found.

SINBAD uses two additional strategies for the Scenario 2, in which data may be missing or disordered, called *quick result calculation* and *noise buffering*.

We explain in detail each of SINBAD's building blocks in the following sections.

3.2 Incremental DBSCAN

One of the main components of SINBAD is the *Incremental DBSCAN* [2] (iDS). It replaces DBSCAN as the algorithm responsible for continuously clustering data arriving from a stream of features. The core of iDS relies on its *state* management, which allows cluster updates to be performed in $O(n)$ while the standard DBSCAN requires $\Theta(n^2)$, where n is the number of features in the feature window. In the following sections, we present iDS and its role in the feature stream processing step.

3.2.1 Overview. The forward and backward steps from SINBAD are similar to those from GCB. The iDS will be executed at every iteration of the forward step, i.e., whenever a new feature is added to the feature window. Likewise, it will also be executed at every iteration of the backward step, i.e., whenever a feature is removed from the feature window after an event is found. Clusters calculated by iDS are identical to DBSCAN's, however, the former is faster: while DBSCAN reprocesses the entire feature window whenever

a feature is added to it or removed from it, iDS only processes part of it. This is possible thanks to the state it maintains, which is composed of the clusters that are currently being processed.

3.2.2 Forward search step. In the forward search step (cf. Section 2.3.1), new features are added, one by one, to the feature window and are afterwards processed by iDS.

Clustering decisions made by iDS depend on the characteristics of its *state*, which contains the mappings between features in the feature window and clusters. Whenever a new feature is available, iDS first analyzes its state before defining how to cluster it.

In the next paragraphs, we describe all possible state configurations that can be found in iDS when a new feature is available in the processing window and the triggered actions.

Let C be the set of all clusters in iDS' state, $C_0 \subset C$, the cluster containing noise features, $\mathcal{W}$ the set of all features in the feature window, f a new feature, and $d(f_1, f_2)$ the euclidean distance between features $f_1 \in \mathcal{W}$ and $f_2 \in \mathcal{W}$.

Case 1) If the window $\mathcal{W}$ is empty, there are no clusters and a new feature f is considered noise. Consequently, f will be part of C_0.

$$When\ C = \emptyset :$$
$$then\ C := C_0 := \{f\}$$

Case 2) If there are previously detected clusters but no noise, i.e., $C_0 = \emptyset$, then, depending on the distance between f and other features present in existing clusters, either f will be considered noise and added to C_0 or it will be added to at least one cluster C_i. If f has a distance smaller than ϵ to at least one feature $c_i \in C_i$, then f is added to C_i. Similarly, if f, at the same time, has a distance smaller than ϵ to features contained in other clusters, they must all be merged together with C_i. Finally, if the distance between f and all other features in $\mathcal{W}$ is greater than ϵ, then f is classified as noise and must be added to C_0.

$When\ C_0 = \emptyset\ and\ |C| > 0 :$
$$\textbf{if}\ \forall c \in \mathcal{W} : d(c, f) > \epsilon,\ \textbf{then}\ C_0 := \{f\}$$
$$\textbf{else}\ C_i := \{f\} \cup \{C_i \mid c_i \in C_i,\ d(c_i, f) \leq \epsilon\}\ and$$
$$C_i := C_i \cup \{C_j \mid C_j \subset C, d(w, f) \leq \epsilon, w \in \mathcal{W},\ w \in C_j\}.$$

Case 3) If the only available cluster in window $\mathcal{W}$ is C_0, then either f will be added to C_0, in case it is classified as noise or it will be added to a new cluster C_1 together with features from C_0 whose distances to f are smaller than or equal to ϵ.

$When\ C = \{C_0\}\ and\ C_0 \neq \emptyset :$
$$\textbf{if}\ \forall c_0 \in C_0 : d(c_0, f) > \epsilon\ ,\ \textbf{then}\ C_0 := C_0 \cup \{f\}.$$
$$\textbf{else}\ C_1 := \{f\} \cup \{c_0 \mid c_0 \in C_0, d(c_0, f) \leq \epsilon\}\ and$$
$$C_0 := C_0 \setminus C_1.$$

Case 4) If there are at least C_0 and C_1 in $\mathcal{W}$, then either (i) f will be classified as noise, (ii) f will compose a new cluster with at least one element of C_0, (iii) clusters having elements with a distance to f less than or equal to ϵ will be merged together and will contain f, or (iv) f creates a cluster as in (ii) and thus combines clusters as in (iii).

$When\ C \neq \emptyset\ and\ |C| > 1 :$
(i) if $\forall w \in W : d(w, f) > \epsilon$ **then** $C_0 = C_0 \cup \{f\}$
(ii) else if $\exists c_0 \in C_0 : d(c_0, f) \leq \epsilon\ and\ \forall c_i \in \bigcup_{i=1}^{|C|-1} C_i : d(c_i, f) > \epsilon$
$$\textbf{then}\ C_n := \{f\} \cup \{c_0 \mid d(c_0, f) \leq \epsilon, c_0 \in C_0\}\ and$$
$$C_0 := C_0 \setminus C_n\ and\ C := C \cup C_n.$$
(iii) else if $\forall c_0 \in C_0 : d(c_0, f) > \epsilon\ and\ \exists c_i \in C_i : d(c_i, f) \leq \epsilon$
$$\textbf{then}\ C_i := \{f\} \cup C_i \cup \{C_j \mid d(f, c_j) \leq \epsilon, c_j \in C_j, C_j \subset C\}$$
(iv) else if $\exists c_0 \in C_0 : d(c_0, f) \leq \epsilon\ and\ \exists c_i \in C_i : d(c_i, f) \leq \epsilon$
$$\textbf{then}\ C_n := \{f\} \cup \{c_0 \mid d(c_0, f) \leq \epsilon, c_0 \in C_0\},$$
$$C_0 := C_0 \setminus C_n\ and\ C := C \cup C_n\ and$$
$$C_i := \{f\} \cup C_i \cup \{C_j \mid d(f, c_j) \leq \epsilon, c_j \in C_j, C_j \subset C\}.$$

3.2.3 Backward search step. During the backward search step (cf. Section 2.3.1), the oldest features from the feature window are removed, one by one. While the standard DBSCAN re-calculates all clusters every time a feature is removed, iDS only updates the clusters affected by the feature removal. During the backward stage iDS has a $O(n)$ time complexity against $\Theta(n^2)$ from standard DBSCAN.

Similarly to the forward stage of iDS, we analyze all possible state configurations at the moment a feature is removed.

Case 1) If f is noise, i.e., $f \in C_0$, then removing f will not affect any other clusters.

$$When\ f \in C_0$$
$$\textbf{then}\ C_0 := C_0 \setminus \{f\}.$$

Case 2) If $f \in C_i$ is not noise, then it is necessary to verify that $C_i \setminus \{f\}$ is still a valid cluster, i.e., whether it has at least 2 features left, and that all of the remaining elements are *reachable*, i.e., there is a path $p(a, b)$ from every $a \in C_i$ to $b \in C_i$, such that the distance between every consecutive feature in the path is smaller than a given ϵ (cf. Section 2.2). Finally, in case f is a *bridge* in the path $p(a, b)$, removing it may break C_i into multiple smaller clusters and/or generate noise features. This situation is solved by regenerating the clusters using only the remaining features from $C_i \setminus \{f\}$.

$When\ f \in C_i, C_i \neq C_0 :$
if $|C_i \setminus \{f\}| < 2$
then $C_i := C_i \setminus \{f\}\ and\ C_0 := C_0 \cup C_i$
else if $\exists a \in C_i,\ \forall b \in \{c_i \mid c_i \in C_i,\ d(f, c_i) \leq \epsilon\}, b \neq a : d(b, a) \leq \epsilon$
then $C_i := C_i \setminus \{f\}$.
else if $a \in C_i \setminus \{f\}, b \in C_i \setminus \{f\} : \nexists p(a, b)$
then perform Case 4 from forward step on each element of C_i added to an initially empty window. The non-noise clusters found, if any, will replace C_i and all noise will be added to C_0.

3.3 Optimizations for Scenario 2

The Scenario 2 is characterized by missing and disordered tuples (cf. Section 2.1). As data is served in batches of 1000 tuples, the processing algorithm cannot immediately differentiate between missing and *late* tuples. The only mechanism available for that is the guarantee that a missing tuple will not be more than 20000 positions away from its expected position, i.e., it is necessary to wait for up to 20 batches of tuples in order to be sure that a measurement tuple is missing or not.

As a consequence of that, there may be situations where missing tuples are served after their corresponding feature is processed. This leads to a loss of accuracy on detected events, since *incomplete features*, i.e., features calculated with incomplete data, may be added to clusters that differ from those to which they would be added if they were *definitive features*, i.e., built with complete data. GCB's solution for that issue is to calculate results using incomplete features despite the accuracy loss. Another solution is to stop the execution of the algorithm and wait for up to 20 batches of data whenever there are missing tuples. However, such an approach would delay the processing of solutions associated to the incomplete feature and subsequent ones.

In the next paragraphs we propose two approaches, *noise buffering* and *quick result calculation*, to avoid halting the entire execution of the algorithm when incomplete features are present.

3.3.1 Noise buffering. Incomplete features directly affect the event detection as they may be clustered to noise features present in the event solution set or be part of the solution set when their definitive version is calculated. In those situations, in order to accurately calculate the result associated to those missing and noise features, it is necessary to wait for the missing tuples to be available.

Noise buffering (NB) keeps track of the points considered noise that depend on missing tuples. It buffers the data in $\mathcal{W}$ so, once the missing tuples are available, the results associated with the points considered noise in the solution set or the incomplete features can be calculated.

3.3.2 Quick result calculation. The *quick result calculation* (QRC) approach targets at calculating results for definitive or incomplete features that can be answered immediately or after a few iterations of the forward search step. For the latter case, QRC is used with NB so the algorithm is not halted while waiting for more features.

A first case where QRC can be applied is when a definitive feature has a distance smaller than or equal to ϵ to another definitive feature. In that situation, both features will never be part of an event, independently of any incomplete features in $\mathcal{W}$, because neither of them will be updated. For those features, it is always possible to *immediately* associate a negative result to them.

QRC can also take advantage of temporal constraints and quickly calculate the result of noise or incomplete features having their ids in the temporal interval of a cluster (cf. Section 2.3.1). Let f be a feature having its id in the temporal interval of a cluster. If f is not noise, then it is be part of a cluster and consequently cannot be an event. If f is noise, then it also cannot be an event because it is "surrounded" temporally-wise by the same cluster and, consequently, cannot be situated between two non overlapping clusters. Hence, when the id of f is in the temporal interval of a cluster, it is enough to determine if f is part of an event or not. We can assert that any feature, incomplete or not, having its id in the temporal interval of a cluster is *not* an event. Notice, however, that it is necessary to wait for the features around the incomplete features to form a cluster before being able to output a negative result.

4 EVALUATION METHODOLOGY

In this section, we discuss the implementation of SINBAD and the environment of the experiments used to analyze its performance.

4.1 Evaluation server

The *evaluation server* interacts with a *client*, which encapsulates a solution implementation defined by a competitor, through an HTTP API. The main responsibilities of the server are (i) answering requests for smart meter tuples from a client, (ii) receiving results calculated by a client, and (iii) sending a performance summary of the results sent by a client.

The HTTP API specifies two calls: a GET request for a batch of data, and a POST request for sending the results of a calculation. Whenever a client issues a GET request, it specifies the context scenario, which can be Scenario 1 or 2. The server answers with a batch of smart meter measurements containing up to 1000 tuples wrapped in JSON. The server does not process parallel requests.

When a client finishes calculating the result associated to a feature, it must send it to the server using a POST request where it defines the id of the feature, a boolean indicating whether the feature is an event or not and the median of the interval of the event (cf. Section 2.3.1).

Finally, when a client requests for data after all batches of the dataset were sent, the server answers with a summary of the execution in terms of percentage of true positives and timeliness.

The Grand Challenge organizers made available a Docker [3] image containing an implementation of the server.

4.2 SINBAD's implementation

SINBAD[4] is implemented in C++14 [5] and uses libcurl [6] to access the evaluation server's API and RapidJSON [7] to parse incoming data and wrap output data. SINBAD is encapsulated by Docker.

We opt to implement SINBAD from scratch instead of using a general purpose stream processing engine such as Apache Flink [8] because (i) such implementation would not benefit from Flink's parallelism since the evaluation server does not process multiple requests at the same time; and (ii) as the algorithms are straightforward, the overhead for developing an efficient and straight-to-the point tailored solution is small. The evaluation server's lack of support for parallel requests is also the reason why SINBAD is essentially sequential. As network access for downloading batches and uploading results is SINBAD's bottleneck (more details in Section 5), there would be almost no performance improvement by

[3] https://www.docker.com
[4] https://github.com/hpi-des-debs2020/hpi-des-debs2020-grand-challenge
[5] https://isocpp.org/wiki/faq/cpp14-language
[6] https://curl.haxx.se/libcurl/
[7] https://rapidjson.org/
[8] https://flink.apache.org/

processing many batches in parallel. Additionally, the code would have to be more complex to manage a parallel SECBED algorithm.

In SINBAD's implementation, we consider a feature to be incomplete when there are more than 100 missing points associated to it. That threshold was chosen after observing that the value of features calculated with 100 or less points where in average only 1.6% smaller or greater than the original value.

4.3 Baselines

We use 3 baselines to analyze the performance of SINBAD: *GCB-Python*, *GCB-C++*, and *SINBAD-NoQRC*.

GCB-Python is the original implementation of GCB made available to all competitors by the organizers of the Grand Challenge. It is implemented in Python and uses DBSCAN from scikit-learn[9] and library requests [10] for dealing with JSON parsing and HTTP requests. GCB-Python does not have a mechanism such as NB (cf. Section 3.3) for buffering data while waiting for missing tuples in Scenario 2. Hence, it always calculates results based only on the available information.

GCB-C++ is our implementation of GCB. It operates exactly as GCB-Python but was developed in C++14 and uses RapidJSON and libcurl. DBSCAN was also implemented in C++14. Similarly to GCB-Python, GCB-C++ cannot wait for missing tuples.

SINBAD-NoQRC is an implementation of SINBAD without the optimization QRC for Scenario 2. It still has implemented NB and consequently can wait for missing tuples before calculating results.

4.4 Metrics

In order to understand the performance of SINBAD, we analyze three metrics: *accuracy*, *runtime*, and *latency*.

Accuracy measures the number of true positives available in the set of solutions sent by a client.

Runtime is the time taken by a client to process all data sent by the evaluation server. It measures the time interval between the moment that the first batch is sent to the client and the moment that the last solution is received by the server.

Latency is the average of *individual latencies*. An individual latency is the time taken by a client to output a result associated to a feature. Each individual latency measures the time interval between the moment that the first batch having tuples associated to a feature is sent and the moment that the result associated to that feature is received.

4.5 Experiment

In our experiments, we evaluate each algorithm, i.e., SINBAD, GCB-Python, GCB-C++, and SINBAD-NoQRC on Scenarios 1 and 2.

Every experiment involves executing an algorithm implementation *three* times in the context of a scenario and calculating the averages of the metrics, i.e., accuracy, runtime and latency. Each execution starts when the evaluation server is ready to receive requests and ends when the evaluation server acknowledges the reception of all results from the algorithm implementation.

The algorithm implementations are encapsulated in Docker containers. Client and server containers are deployed on the same machine, which is equipped with an Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz having 2 threads per core and 32G RAM.

[9]https://scikit-learn.org
[10]https://requests.readthedocs.io/en/master/

5 PERFORMANCE ANALYSIS

We split the performance analysis of SINBAD in three parts. We analyze the results of the experiments in the context of Scenarios 1 and 2, and, then, we profile SINBAD's processing times.

5.1 Scenario 1

We summarize the metrics of experiments of the Scenario 1 in Table 1. SINBAD's runtime is more than 5 times shorter than GCB-Python's and its latency is more than 7 times smaller than GCB-Python's. On the other hand, the performance differences between SINBAD and GCB-C++ are much smaller: 11 seconds for runtime and around 500ms for latency.

The runtime and latency differences between SINBAD and GCB-C++ are mainly explained by the performance of iDS against DBSCAN. The impact of our choice of libraries and programming language causes the difference between GCB-C++ and GCB-Python performances. As all solutions are based on SECBED and there are no missing tuples in Scenario 1, the accuracy of all experiments is 100%. For the same reason, SINBAD and SINBAD-NoQRC have very similar metrics.

Table 1: Runtime, latency, and accuracy in Scenario 1.

	runtime	latency	accuracy
SINBAD	572 s	22.90 ms	100%
SINBAD-NoQRC	572 s	22.87 ms	100%
GCB-C++	581 s	23.44 ms	100%
GCB-Python	2953 s	165.32 ms	100%

5.2 Scenario 2

We show the experiment metrics of Scenario 2 in Table 2. The runtimes of the experiments are similar to the results in the experiments for the Scenario 1 shown in Table 1. However, there is a substantial difference among the values of latency and accuracy.

Table 2: Runtime, latency, and accuracy in Scenario 2.

	runtime	latency	accuracy
SINBAD	576 s	58.34 ms	81.4%
SINBAD-NoQRC	574 s	149.1 ms	81.4%
GCB-C++	601 s	24.53 ms	64.5%
GCB-Python	21,930 s	1,231 ms	64.5%

SINBAD has a latency around 2.5 times smaller than SINBAD-NoQRC and more than 21 times smaller than GCB-Python. The difference between SINBAD and SINBAD-NoQRC describes the impact of the QRC and NB optimizations on SINBAD. Notice that SINBAD's latency is around 2.3 times greater than GCB-C++'s. At the same time, SINBAD's accuracy is around 26% greater than GCB-C++'s. That happens because of an accuracy vs. latency *trade-off*.

Whenever there are data missing, the event detection algorithm must decide between calculating a result associated to an incomplete feature or waiting for up to 20 batches to have all necessary data to calculate the result. That trade-off is illustrated in Figure 1, where we variate the maximum number of batches to wait before calculating a result associated to an incomplete feature. In order to improve the accuracy from around 60% to up to around 70%, it is necessary to increase the latency by a factor of more than two.

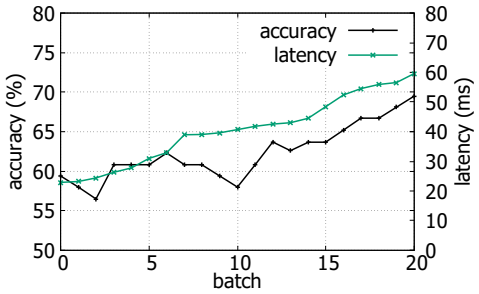

Figure 1: Trade-off between accuracy and latency for a SINBAD execution in Scenario 2.

While it is not evaluated in the Grand Challenge, SINBAD also has a good performance for true negatives, as shown in Table 3. Notice that SINBAD's F1 score is 0.80, i.e., around 48% greater than GCB-C++'s. We omit SINBAD-NoQRC and GCB-Python as their accuracies are identical to SINBAD and GCB-C++, respectively.

Table 3: F1 score, precision, and recall for Scenario 2.

	F1 score	Precision	Recall
SINBAD	0.80	0.81	0.78
GCB-C++	0.54	0.64	0.46

5.3 SINBAD profiling

We divide the entire execution process of each experiment into four steps and measure the amount of time spent in each one of them. *Step 1* refers to asking the server for data and downloading them; *Step 2* refers to parsing JSON data received from the server; *Step 3* refers to the actual event-detection data processing using SINBAD or one of the baselines; and *Step 4* refers to uploading the results back to the server. Figure 2 illustrates the execution time per step, for Scenario 2. We choose to omit results from Scenario 1 because they are similar to Scenario 2's. As GCB-Python has a performance one order of magnitude greater than other algorithms, we also omit it for the benefit of readability.

The first important characteristic to notice in Figure 2 is the impact of networking on the time consumption of Steps 1 and 4, which is slightly the same for all algorithms and is up to around 88% of the execution time. We also observe that the impact of the JSON library on Step 2 of all implementations responds for around 11% of the execution time, leaving around 0.5% for Step 3.

In comparison to GCB-C++, in particular, Step 3 shows SINBAD's improvement in terms of execution time. It also shows that all approaches are inefficient in terms of networking (cf. Steps 1 and 4). This happens, mainly, because the evaluation server does not allow requests to be processed in parallel.

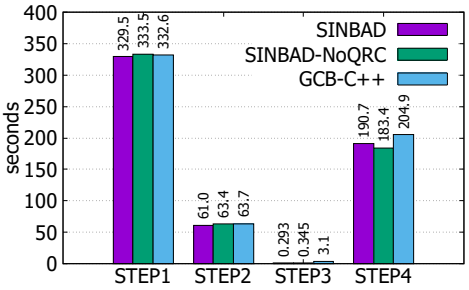

Figure 2: Breakdown of execution time in Scenario 2.

6 RELATED WORK

SINBAD, the proposed solution for the Grand Challenge, is built on top of existing algorithms and approaches from the literature with the objective of improving execution time, latency and accuracy metrics of the GCB. SINBAD has mainly 3 building blocks, the event detection algorithm SECBED, the clustering algorithm iDS and the Scenario 2 optimizations QRC and NB.

We employ SECBED almost as it is defined in [1]. The only important modification is the replacement of DBSCAN by iDS and the incorporation of the optimizations for Scenario 2, QRC and NB.

We based iDS on the incremental version of DBSCAN defined in [2] and only adapted it to fit SECBED characteristics and the optimizations of Scenario 2.

The optimizations QRC and NB are very specific for solving the Grand Challenge and use classic techniques of concurrent and parallel processing.

7 CONCLUSION

In this work, we present SINBAD, an algorithm for detecting events on streamings of smart meter measurements. Combining iDS, a fast clustering algorithm, QRC and NB, two scenario specific optimizations, and an efficient implementation, SINBAD manages to outrun all baselines on the evaluated datasets. In the Scenario 1, SINBAD has the same accuracy as the other baselines but is five times faster than GCB-Python and 9 seconds faster than GCB-C++. In Scenario 2, it has an accuracy of 81.4% and a F1 score of 0.80 which are 26% and 48% better than GCB-C++'s, respectively.

ACKNOWLEDGMENTS

This research was partially funded by the HPI Research School on Data Science and Engineering.

REFERENCES

[1] Karim Said Barsim and Bin Yang. 2016. Sequential Clustering-Based Event Detection for Non-Intrusive Load Monitoring. *Computer Science and Information Technology.*

[2] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. 1998. Incremental Clustering for Mining in a Data Warehousing Environment. In *Proceedings of the 24rd International Conference on Very Large Data Bases.*

[3] 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.*

[4] 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).*