Grand Challenge: Real-time High Performance Anomaly Detection over Data Streams

Dimitrije Jankov, Sourav Sikdar, Rohan Mukherjee, Kia Teymourian and Chris Jermaine
Rice University
dj16,ss107,rm38,kiat,cmj4@rice.edu

ABSTRACT

Real-time analytics over data streams are crucial for a wide range of use cases in industry and research. Today's sensor systems can produce high throughput data streams that have to be analyzed in real-time. One important analytic task is anomaly or outlier detection from the streaming data. In many industry applications, sensing devices produce a data stream that can be monitored to know the correct operation of industry devices and consequently avoid damages by triggering reactions in real-time.

While anomaly detection is a well-studied topic in data mining, the real-time high-performance anomaly detection from big data streams require special studies and well-optimized implementation. This paper presents our implementation of a real-time anomaly detection system over data streams. We outline details of our two separate implementations using the Java and C++ programming languages, and provide technical details about the data processing pipelines. We report experimental results and describe performance tuning strategies.

CCS CONCEPTS

• **Information systems → Data streams**; **Data stream mining**;

KEYWORDS

Data Stream Processing, Data Stream Mining

ACM Reference format:
Dimitrije Jankov, Sourav Sikdar, Rohan Mukherjee, Kia Teymourian and Chris Jermaine. 2017. Grand Challenge: Real-time High Performance Anomaly Detection over Data Streams. In *Proceedings of DEBS '17, Barcelona, Spain, June 19-23, 2017,* 6 pages.
https://doi.org/10.1145/3093742.3095102

1 INTRODUCTION

There has been a growing demand for efficient data stream processing systems in recent years. Consequently, academic and industrial attention has been focused on improving existing approaches for high throughput data stream and event processing solutions. The ACM International Conference on Distributed and Event-based Systems (DEBS) includes a Grand Challenge series that provides a competition aimed at both research and industrial event-based systems.

DEBS '17, June 19-23, 2017, Barcelona, Spain

ACM ISBN 978-1-4503-5065-5/17/06.
https://doi.org/10.1145/3093742.3095102

The overall goal of 2017 DEBS Grand Challenge [3] is to implement efficient high-performance anomaly detection system for sensor data streams generated by manufacturing equipment. Abnormal behavior of a manufacturing machine should be detected based on a measurement data stream generated by its sensors. The organizers of the DEBS grand challenge propose a specific anomaly detection method where the data produced by each sensor is clustered and the state transitions between the observed clusters are modeled as a Markov chain. Anomalies are detected as sequences of transitions that happen with a probability lower than a given threshold.

Recent outlier analysis algorithms [1, 2] use techniques like nearest neighbor, clustering, spectral techniques, classification, particle filter, etc for this task. In recent years novel anomaly detection framework [4] are proposed that can be used to detect anomalies in a streaming fashion by making only one pass over the streaming data. The organizers of the DEBS Challenge specify to use a combination technique of clustering and Markov chain in a data stream for the challenge implementation, to simplify and unify the results of challenge participants.

In this paper, we:

- Provide a brief description of the implementation objectives of the DEBS 2017 Grand Challenge (Section 3). For more details about the DEBS Grand Challenge, we refer the reader to the description paper provided by the organizers [3] and the DEBS 2017 conference website [1].
- Describe a system architecture for a customized data stream processing system for anomaly detection. The processing pipeline and system architecture are described in Section 4.
- Provide implementation details we used to improve the stream processing throughput and latencies in our Java and C++ implementations. We focus on the methods that we used to avoid unnecessary data processing.
- Evaluate our solutions, showing that their throughput and processing latency. Evaluation makes up Section 6 of the paper.

2 CHALLENGE DATA

The data stream is generated by two types of manufacturing machines: (1) injection molding machines and (2) assembly machines. Injection molding machines are equipped with sensors that measure various parameters of a production process: distance, pressure, time, frequency, volume, temperature, speed and force. All the measurements taken at a certain point in time result in a 120-dimensional vector consisting of values of different types.

Figure 1 depicts the raw stream produced by a molding machine. The sensor values vary based on the state of a molding machine.

[1] http://www.debs2017.org/call-for-grand-challenge-solutions/

For example, the stream normally goes from a range of (1700-1900) to a range of (2200-2500) and returns to the previous range of (1700-1900). In the case of any abnormal behavior of the machine, the values are most likely out of those ranges. In Figure 1, the data points with values more than 8000 are likely anomalies.

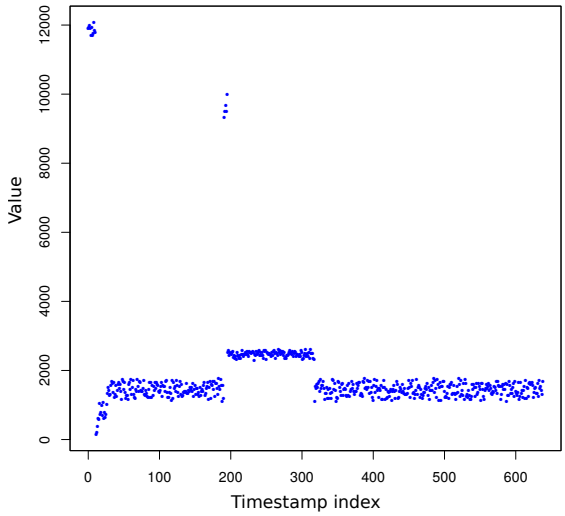

Figure 1: Raw Sensor Data Output Stream - Values in different ranges based on different states of the molding machine.

The data set also includes meta-data about the machines and measurement data (streaming sensor data). Meta-data includes information about the machine type, the number of sensors per machine and the number of clusters that is required for anomaly detection. Each measurement (also referred to as an observation) is timestamped and it is described using the OWL ontology (Web Ontology Language)[2].

The input data stream is in RDF[3] format which is a standard model for data interchange. The data messages are formatted using RDF Turtle notation[4] to enable streaming of RDF statements. This is because in RDF Turtle format each text line serializes a single RDF statement (an RDF triple (subject, predicate, object)).

3 ANOMALY DETECTION METHOD

3.1 Workflow

The DEBS grand challenge describes an anomaly detection method based on a 3 stage data processing pipeline. The parameters for the data processing pipeline are shown in Table 1. Before the data is fed to the processing pipeline, a sliding window over the data stream generates windows of size W. On each arrival of a new observation, a new data window will be generated by dropping the first (oldest) data value in the window and adding the new one. The windows ere then processed by the following three stages:

1. Finding Clusters. The first step clusters the elements in the window using the K-Means method. The grand challenge specifies that for each dimension, our system has to maintain K cluster centers, using the numbers 1 to K as seeds for the initial K centroids. The number K is defined in the meta-data for each dimension of each individual machine.

All data items inside the window should be used to compute the cluster centroids. The initial cluster centers are the first K distinct items inside the window, starting from oldest timestamp. On each shift of the window, the cluster centers should be recomputed again, using the first K distinct values to seed the new clustering. The clustering algorithm must compute M iterations to cluster the data inside a given data window, unless it terminates earlier. The default value of M can be obtained from the meta-data specifications. In the case that a given window has less than K distinct values, then the number of clusters is equal to the number of distinct values inside that window. If a data point has the same distance to more than one centroids, it must be associated with the cluster that has the highest centroid value.

2. Training the Markov Model. Using the cluster assignments from step 1, a Markov Model is trained on the current window. The Markov Model consists of a probability matrix of transition probabilities between states. Following the first-order Markov property, it is assumed that a state transition is solely dependent on the last state visited. The states correspond to the different clusters obtained from the K-Means step.

To train the Markov Model, the number of transitions from one state to another in the window are counted. The counts are then normalized and used to assigns probability value to each element in the transition matrix.

The trained Markov model is used only in the current time window. When a new data point is processed, the first data point in the window will be dropped, the clustering step recomputed, and a new Markov Model trained.

3. Finding Anomalies. For each data window, the last N data points are considered as possible anomalies. If the sequence of last N data points has a combined transition probability lower than a threshold T_d, an alert about the machine is raised. When an anomaly is detected, the first time stamp amongst the last N data points is output as an alert that an anomaly has been detected at that instant of time.

Parameter Name	Description
W	Window size
N	Number of state transitions
M	Maximum number of iterations for K-Means algorithm
T_d	Probability threshold for anomaly detection

Table 1: System parameters.

[2]Web Ontology Language https://www.w3.org/OWL/
[3]Resource Description Framework (RDF) https://www.w3.org/RDF/
[4]Turtle, the Terse RDF Triple Language https://www.w3.org/TR/turtle/

3.2 Example

We present the entire workflow of the data processing pipeline below with an example. Assume that the current window of observations is:

$$[9.42, 9.64, 8.87, 9.38, 8.42, 9.6, 9.27, 9.55, 9.26, 9.62]$$

The number of cluster centers is specified as 4. The initial cluster centers are determined as the first 4 distinct values for that observation window as $[9.42, 9.64, 8.87, 9.38]$. After convergence of the K-Means clustering algorithm, the vector of cluster assignments for the given observation window is $[0, 1, 2, 0, 2, 1, 3, 1, 3, 1]$ and the learned clusters are:

```
Cluster 0: Centroid: 9.4
Observations: [9.42, 9.38]

Cluster 1: Centroid: 9.6025
Observations: [9.64, 9.6, 9.55, 9.62]

Cluster 2: Centroid: 8.645
Observations: [8.87, 8.42]

Cluster 3: Centroid: 9.265
Observations: [9.27, 9.26]
```

The matrix of transition probabilities is then:

$$\begin{bmatrix} 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 1/3 & 2/3 \\ 1/2 & 1/2 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

The threshold probability for the last 5 transition is calculated as follows: $(1/2) \times (2/3) \times 1 \times (2/3) \times 1 = 0.222$

3.3 Challenge Characteristics

The DEBS Challenge [3] defines the following specific characteristics:

- **Data.** Each input data message includes measurements from some subset of the sensors from some subset of the machines. The system should memorize the past data stream to generate the sliding data windows.
- **Processing System Elasticity.** manufacturing machines can join and leave the data stream generation so that the processing system should be elastic to react to the higher throughput of the raw data stream.
- **Real-Time Processing.** The input data stream has to be processed with low latency to achieve the real-time deadline constrain. Any results that are output after a deadline are unacceptable. This constraint causes some challenges for the processing pipeline because that data items cannot be queued for long inside the processing pipeline. Also the parsing of the input data should be highly efficient.
- **Sorted Output.** The output of anomalies should be sorted based on timestamp of the anomaly. Also, the output messages require a sequential timestamp index number.

4 SYSTEM ARCHITECTURE

We designed the system architecture of our data stream processing system based upon the general stream processing systems architecture described by Stonebraker et al. [5]. The single design principle that we followed throughout system design was to keep the data streaming throughout the processing pipeline—to achieve real-time processing, we do not cache or index the data.

Figure 2 depicts the system architecture, consisting of the following 3 main components:

(1) The **Input or Entry Component** responsible for receiving the raw data stream from messaging queue system (RabbitMQ[5]), parsing the text message, extracting the RDF statements and generating the data windows.
(2) The **Anomaly Detection Component** responsible for executing the anomaly detection tasks over a given data window.
(3) The **Output Component** responsible for message ID generation, formatting, and sending the final anomaly report message as a set of RDF statements to the output message broker.

In the following subsections we describe each of the components.

4.1 Input Component

The input component reads the messages from the message broker and sends them to the RDF parser. Each message includes a single timestamped observation group. It includes multiple measurements described by multiple RDF statements.

The initial task of the input component is to parse the plain RDF text messages. After parsing, the input component passes the parsed input to a data window manager that is responsible for building the data windows. It maintains a memory of the previously observed measurements from from each specific machine. Each timestamped message (also named an *observation group* includes measurements from some of the machines and some of its dimensions. This requires that the input component keeps the past window for each machine dimension in memory to build the next sliding window when a new data measurement arrives in the data stream. The next step is to dispatch the data windows to the next component for executing anomaly detection. In the case of multi-threaded processing or distribution across multiple worker machines, the window data can be deep copied and dispatched to the down-stream component.

4.2 Anomaly Detection Component

The anomaly detection consists of two steps: (1) clustering the data inside the data window, and (2) calculating the Markov Chain transition matrix and transition threshold. The most important architectural aspect of this component is that all of its computations are functional, so that each can easily be parallelized by using multiple threads or processes. The dispatching of each window in order to parallelize computation tasks is done based using a combination of machine identifier and dimension identifier. Further, each thread used by the anomaly detection component caches the past data window so that the K-Means clustering computation can benefit by starting from previously known centroids.

[5]RabbitMQ is an open source message broker system https://www.rabbitmq.com/

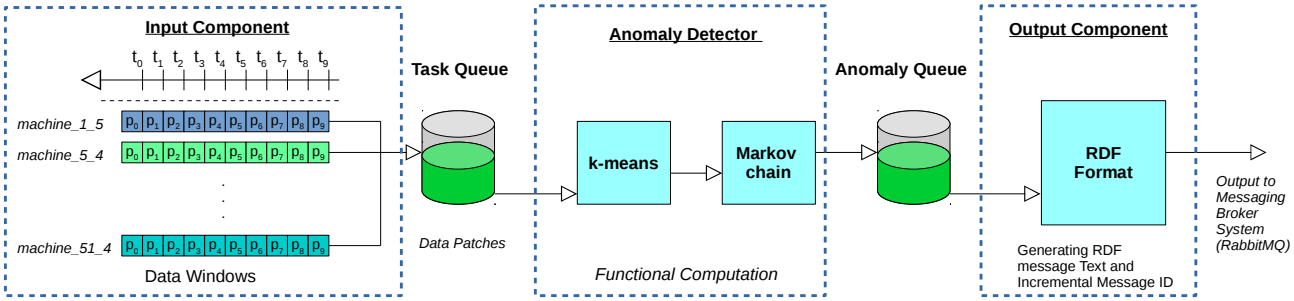

Figure 2: Architecture of our Anomaly Detection System.

4.3 Output Component

In our processing pipeline, the result of the anomaly detection component is forwarded to the output component for reporting.

4.4 Parallelization and Distribution

Figure 3 depicts our architecture for parallelization and distribution of our stream processing pipeline. The processing pipeline is parallelized with two important goals: (1) achieving low processing latency, and (2) achieving higher throughput and higher scalability in the case that input stream is high. Clearly, there are trade-offs optimization between these two factors. Using multiple worker threads (or processes) should not increase the processing latency of a single data item. Hence the data should be queued in the pipeline as quick as possible to achieve real-time processing.

5 IMPLEMENTATION DETAILS

In this section we describe some of the implementation details, and methods we used to tune performance.

We have implemented our solution in C++ [6] and Java[7].

Improved Message Parsing Performance. Our system reads the messages from the message broker (RabbitMQ) as raw bytes which contain RDF as plan text. The system must parse the data and extract measurement values.

We do not use existing RDF libraries such as Apache Jena [8] or OpenRDF[9] to parse the messages because these libraries are designed handle stored RDF data, and not RDF streams. We also have not used any of the existing systems for RDF stream processing such C-SPARQL [6] because those systems are suitable mostly for running continuous RDF queries over data streams, and not running data mining algorithms on an RDF stream.

Instead, we have implemented our own customized RDF parser that can parse the RDF statements with high performance using regular expressions over the plain text data. As the data is formatted in UTF-8 Unicode, we map it to character arrays, and parse the data by by searching for specific patterns using regular expression search. In our Java implementation, we use the following code to map the bytes to a `CharBuffer`:

```
ByteBuffer byteBuffer=ByteBuffer.wrap(bytes);
Charset utf8= harset.forName("UTF-8");
CharBuffer charBuffer=utf8.decode(byteBuffer);
```

The DEBS 2017 Challenge specifies that each message includes an observation group which includes a header part followed by a main body part. The header part includes RDF statements about the a group of measurements, like the timestamp for all following measurement statements in the main body part. We first parse the header part and then use multiple threads to read from different positions of the character buffer and parse the main body part. By apply this method, we can very quickly bypass the repeating RDF statements, such as `rdf:type` statements.

For example we use the following code in Java to search for the start position of specific RDF parts.

```
int findChar(CharBuffer chars, char c, int
    start){
 while (chars.charAt(start) != c) {
  start++;
 }
 return start; }
```

After the parsing step, we send the data to the window generator component that generates the sliding windows on each new data time for the specific (manufacturing machine, dimension) combination. In the next step, the window data (now vectors of double values) are passed to the anomaly detector component.

Metadata preprocessing and efficient storage. A metadata set is provided that describes some metadata values—whether the specific dimension of injection/modeling machine is stateful, what the target number of clusters is, and what is the anomaly threshold for each machine-dimension. The metadata RDF plain text file of size 72.6 MB. As the metadata will not be changed during the stream processing run time, we have pre-processed the file and stored the content in a efficient hand coded serialization stored in 448 bytes. The file is added to our Docker[10] images so that our system can read small file, keep the content in main memory to achieve a a fast start.

After the pre-processing step, we have the following two maps in the main memory. The key of the map is machine id number as integer, and value is an array of integer that includes the cluster

[6]Source Code of our C++ implementation is available at https://github.com/dimitrijejankov/debs-conference
[7]Our Java implementation at https://github.com/kiat/debs2017
[8]An open source library for RDF data processing http://jena.apache.org/index.html
[9]http://rdf4j.org/
[10]https://www.docker.com/

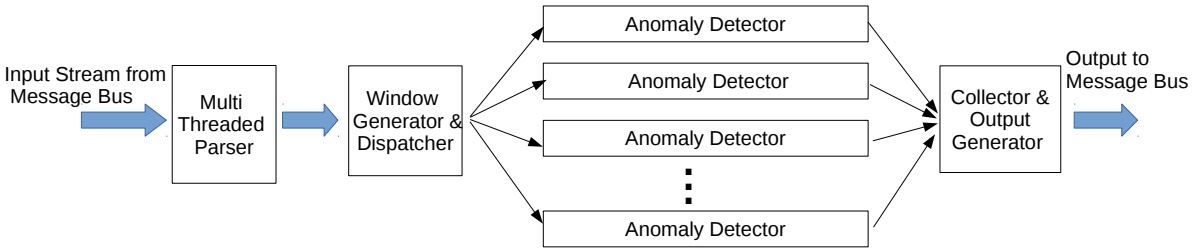

Figure 3: Architecture of our Multi-Threaded/Multi-Processed Anomaly Detection System.

numbers or an array of double for the threshold numbers. The array index is the dimension number. We have initials the array only for th stateful dimensions to reduce the data size in RAM.

```
HashMap<Integer, int[]> clusterNos ;
HashMap<Integer, double[]> thresholds;
```

In our Java implementation we use the following hand-coded serialization approach to serials the data for writing it to a file. We selected the hand-coded serialization to speed up the reading of metadata. Our serialization approach includes 2 main steps, first we write the size of the map, then for each key value pair, first write the key and then loop through the vales and write them. Our deserialization is the inverse run of this method.

```
// write first the total size of the map.
ByteBuffer byteBufferSize =
    ByteBuffer.allocate(4);
byteBufferSize.putInt(clusterNo.size());
// write to file.

For_each_key_value_pair_do{
ByteBuffer byteBuffer =
  ByteBuffer.allocate(4+SensorNo*8);
  byteBuffer.putInt(pair.getKey());
  // Serialize the Array
  for(int i = 0; i < m_Array.length; i++){
   byteBuffer.putInt(m_Array[i]);
  }
// write to file.
}
```

Avoiding the processing of every message. As the main task is to detect anomalies on a vector of doubles, we thought about special cases where the data should not be sent to the anomaly detector component. One cases is when there are less than 3 distinct values inside the window. In such a case there is no anomaly in the window, and we can avoid the computations for cluster assignments and Markov Chain calculations. We observed that in the case of smaller window sizes (size ten or less), in %40 of the cases, the number of distinct values are less than 3 because the sensors are stateful, producing data within a certain range for a long time before they change to the next state, as shown in Figure 1.

Parallelized K-Means for Streaming Data. We have parallelized the cluster assignment step in K-Means clustering by using OpenMP and use multiple threads to execute it. By applying this approach we could achieve some speed up. However, because the grand challenge specifies correctness tests that has to be matched and which depend on the initialization of of the clustering, we could not use apply further modifications of the K-Means algorithm. In the case of larger stream window size, the clustering algorithm could benefit by initializing the clustering process from the previously assigned cluster centers.However, the grand challenge specification disallows this.

In order to speed up the processing and use multiple threads we send multiple windows at the same time to the anomaly detection component. The Input component keeps a list of all parsed data and generated windows data. If it has enough window data for a specific dimension, then it sends list of windows data with a specified size to the anomaly detection thread. The following pseudo code illustrate this for our C++ implementation:

```
unsent_windows : list
update_window_list(w : window)
 unsent_windows.add(w)
            if(unsent_windows.size ==
                SIZE_LIMIT)
   send_to_worker_thread(w);
   unsent_windows.clear()
```

The anomaly detector component receives the list of windows data and processes it in parallel. In the case of anomaly detection, they will be outputted in order of their respective windows position in the list (original raw data timestamp order). The pseudo code illustrate our usage of OpenMP:

```
process_list(send_windows : list)
#pragma omp parallel for ordered schedule(
    dynamic) num_threads(NUMBER_OF_KMEANS)

for each w : window
  anomaly <- perform_anomaly_detection(w)

 #pragma omp ordered
 if(anomaly.dectected)
  output_anomaly(anomaly)
```

Stream Processing. In our Java implementation, we used some of the Java 8 stream processing features, like the following code to parallelized the anomaly detection component.

```
List<WindowData> windows_data;
windows_data.parallelStream()
        .forEach(dataBatch ->
                detectAnomaly(dataBatch));
```

We first buffer a set of data windows inside a list, parallelize it, and then for each of the windows execute the anomaly detection process using multiple threads. The size of the buffered list of widows data is dynamically set based on the window size, so that window size is kept small, so as to not queue the data, and achieve real-time processing. This approach is important to reduce the latency and do not delay the processing of a specific data item too long.

6 PERFORMANCE EVALUATIONS

The DEBS 2017 challenge uses the Hobbit project[11] benchmark platform [12] to check for correctness and measure performance.

In addition, we have conducted performance evaluations on a single server (Linux Ubuntu 16, 4 Cores 3.50GHz, 16GB RAM) for the purpose of performance tuning and finding the bottleneck components in our stream processing pipeline. For our evaluation we used the data set provided by the DEBS 2017 challenge.

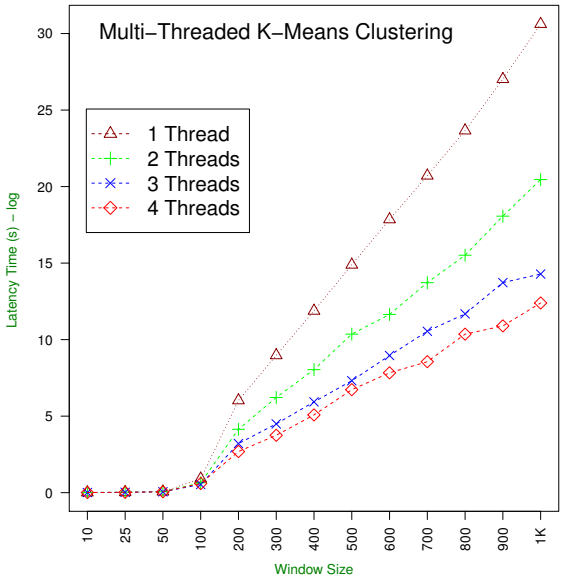

Figure 4: Multi-Threaded Processing of K-Means Clustering - Time of running 1000x K-Means

Performance of components. We have conducted separate performance evaluations of our three components by reading the data from a text file and sending it directly to them and measure the latency times. We send size 10,000 observation groups and measure the average latencies for the separate components, excluding any additional times such as reading the text file from disk.

For the window size of ten, the average results of three runs of our C++ implementation for parsing a complete observation group containing an average of 117 measurements is 122.4 microseconds. The time for anomaly detection (K-Means) for each data window (window size 10) is 5.8 microseconds. Considering that each of the messages generates a new data window, the time to process the whole message will be then in average $117 \times 5.8 = 678.6$ microseconds. The time for generating the output anomaly message is 35.8 microseconds. As we expected, the bottleneck in the processing pipeline is the anomaly detection component, which is in this case six times slower than the input parser component.

For the window of 100, the latency of anomaly detection component increases to 909 microseconds $\times$ 117 numbers = 106,388 microseconds, and it will be 60x slower than the input component.

Since we expected that the anomaly computation is the bottleneck of our processing pipeline, we designed our system to parallelize this component, as shown in Figure 3.

Performance of multi-threaded anomaly detection, and relation to window size. Figure 4 depicts our results for using up to four threads as a function of window size. As the figure shows, using four threads can reduce the latency on larger windows sizes from 30 seconds to 12 seconds for running the anomaly detection component 1,000 times. On the smaller window size (smaller than 100), the latency is almost the same, and it is some times higher when using more threads because of the performance lost caused by synchronization tasks. As our experiment was done on a single machine with four CPU cores, we did not expect our software to scale linearly when the window size increases.

7 CONCLUSION

In this paper, we presented the design and implementation of a custom-built stream processing engine for an anomaly detection task. Our main focus was implementing a stream processing pipeline that could execute computationally-intensive tasks concurrently in parallel, and in real-time.

Acknowledgments. Material in this paper has been supported by the NSF under grant nos. 1355998 and 1409543, and by the DARPA MUSE program.

REFERENCES

[1] Charu C. Aggarwal. 2013. *Outlier Analysis.* Springer Publishing Company, Incorporated.

[2] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (July 2009), 58 pages. https://doi.org/10.1145/1541880.1541882

[3] Vincenzo Gulisano, Zbigniew Jerzak, Roman Katerinenko, Martin Strohbach, and Holger Ziekow. 2017. The DEBS 2017 grand challenge. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS '17, Barcelona, Spain, June 19 - 23, 2017.*

[4] Hao Huang and Shiva Prasad Kasiviswanathan. 2015. Streaming Anomaly Detection Using Randomized Matrix Sketching. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 192–203. https://doi.org/10.14778/2850583.2850593

[5] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. 2005. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.* 34, 4 (Dec. 2005), 42–47. https://doi.org/10.1145/1107499.1107504

[6] Ying Zhang, Minh-Duc Pham, Óscar Corcho, and Jean-Paul Calbimonte. 2012. SRBench: A Streaming RDF/SPARQL Benchmark. In *The Semantic Web - ISWC 2012 - 11th International Semantic Web Conference, Boston, MA, USA, November 11-15, 2012, Proceedings, Part I*, Vol. 7649. Springer, 641–657. https://doi.org/10.1007/978-3-642-35176-1_40

[11] Hobbit Project https://master.project-hobbit.eu

[12] A fully dockerized benchmark platform https://master.project-hobbit.eu