FlinkMan: Anomaly Detection in Manufacturing Equipment with Apache Flink: Grand Challenge

Yann Busnel, Nicolo Riveei, Avigdor Gal

Grand Challenge: FlinkMan – Anomaly Detection in Manufacturing Equipment with Apache Flink

Nicolo Rivetti
Technion - Israel Institute of Technology
nrivetti@technion.ac.il

Yann Busnel
IMT Atlantique / IRISA / UBL
yann.busnel@imt-atlantique.fr

Avigdor Gal
Technion - Israel Institute of Technology
avigal@technion.ac.il

ABSTRACT

We present a (soft) real-time event-based anomaly detection application for manufacturing equipment, built on top of the general purpose stream processing framework Apache Flink. The anomaly detection involves multiple CPUs and/or memory intensive tasks, such as clustering on large time-based window and parsing input data in RDF-format. The main goal is to reduce end-to-end latencies, while handling high input throughput and still provide exact results. Given a truly distributed setting, this challenge also entails careful task and/or data parallelization and balancing. We propose FlinkMan, a system that offers a generic and efficient solution, which maximizes the usage of available cores and balances the load among them. We illustrates the accuracy and efficiency of FlinkMan, over a 3-step pipelined data stream analysis, that includes clustering, modeling and querying.

CCS CONCEPTS

•**Information systems** →*Stream management;* •**Theory of computation** →*Distributed algorithms; Unsupervised learning and clustering;*

KEYWORDS

Anomaly Detection, Stream Processing, Clustering, Markov Chains, Linked-Data

1 INTRODUCTION

Stream processing management system (SPMS) and/or Complex Event Processing (CEP) systems gain momentum In performing analytics on continuous data streams. Their ability to achieve sub-second latencies, coupled with their scalability, makes them the preferred choice for many big data companies. Supporting this trend, since 2011, the ACM International Conference on Distributed Event-based Systems (DEBS) launched the Grand Challenge series to increase the focus on these systems as well as provide common benchmarks to evaluate and compare them. The ACM DEBS 2017 Grand Challenge focuses on (soft) real-time anomaly detection in manufacturing equipment [4]. To handle continuous monitoring, each machine is fitted with a vast array of sensors, either digital or analog. These sensors provide periodic measurements, which are sent to a monitoring base station. The latter receives then a large collection of observations. Analyzing in an efficient and accurate way, this very-high-rate – and potentially massive – stream of events is the core of the Grand Challenge. Although, the analysis of a massive amount of sensor reading requires an on-line analytics pipeline that deals with linked-data, clustering as well as a Markov model training and querying.

The FlinkMan system proposes a solution to the 2017 Grand Challenge, making use of a publicly available streaming engine and thus offering a generic solution that is not specially tailored for this or that challenge. We offer an efficient solution that maximally utilizes available cores, balances the load among the cores, and avoids to the extent possible tasks such as garbage collection that are only indirectly related to the task at hand.

This rest of the paper is organized as follows. Section 2 presents the query engine pipeline, the data set and the evaluation platform, that are provided for this challenge. Section 3 introduces the general architecture of our solution and its rationale. Finally, Section 4 provides details of the implementation as well as the optimizations included in our solution.

2 PROBLEM STATEMENT

The overall goal is to detect anomalies in manufacturing machines based on a stream of measurements produced by the sensors embedded into the monitored equipments. The events produced by each sensor are clustered and the state transitions between the clusters are used to train a Markov model. In turn, the produced Markov model is used to detect anomalies. A sequence of transitions that follows a low probability path in the Markov chain is considered as abnormal, and is flagged as an anomaly.

2.1 Query

The anomaly detection analysis can be modeled as a pipeline with three stages: (**i**) clustering, (**ii**) Markov model training and (**iii**) Markov model querying (*i.e.,* output transition sequences with low probability). These three steps are executed continuously on a time-based sliding window and the whole pipeline is performed independently for each sensor of each machine. The query has 6 parameters: the time-based sliding window size W (in seconds), the initial number of clusters k (non uniform among sensors), the maximum number of iterations of the clustering algorithm M (if convergence has not been reached), the clustering algorithm convergence distance μ, the length of the Markov model path we consider for computing the anomaly probability N, and the probability threshold T below which the path is classified as anomaly. Each event goes through all the mentioned stages so that a single event may

DEBS'17, Barcelona, Spain

DOI: 10.1145/3093742.3095099

Event	Physical Timestamp	Logical Timestamp	Value
r_0 = (	1485903716000,	1155,	−0.04)
r_1 = (	1485903717000,	1165,	−0.04)
r_2 = (	1485903718000,	1175,	+0.02)
r_3 = (	1485903719000,	1185,	−0.0)
r_4 = (	1485903720000,	1195,	−0.01)
r_5 = (	1485903721000,	1205,	−0.04)
r_6 = (	1485903722000,	1215,	+0.0)
r_7 = (	1485903723000,	1225,	−0.02)
r_8 = (	1485903724000,	1235,	+0.0)
r_9 = (	1485903725000,	1245,	+0.02)

Table 1: Example of an input window of size $W = 10$.

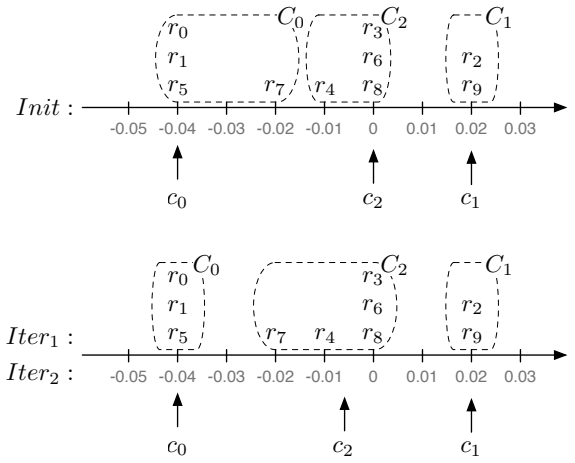


Figure 1: Clustering (k-means) with $k = 3$.

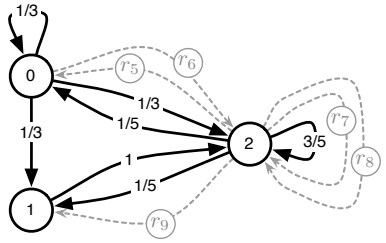
change the clustering, modify the Markov model, and trigger an anomaly detection. It is worth noting that the detected anomalies must be ordered with respect to the ordering in the input stream.

Example 2.1. Table 1 contains an input windows of size $W = 10$.

Clustering First, the clustering algorithm groups all readings (from r_0 to r_9) into $k = 3$ clusters. To do so, a k-means algorithm is initialized: the cluster centers are set to the k first values encountered (represented as c_0, c_1 and c_2 in the *Init* part of Figure 1). Then, a first grouping is produced according to these centers. Several iterations are then launched, until convergence, to find the best-fitted clustering. In our example, after a the third iteration ($Iter_2$ in Figure 1), an equilibrium is reached and the clusters, represented in the bottom part of Figure 1, are returned.

Model training Then, based on this history, a trained Markov chain is computed (Figure 2). This Markov model illustrates, for instance, that the probability is 1/3 to move from cluster C_0 to cluster C_2 (respectively states 0 and 2 in the Markov chain), and is 0 to move to C_0 from C_1.

Model quering Finally, the path represented by the last 5 readings raises an anomaly. In fact, as demonstrated in the bottom of Figure 1, r_4 belongs to C_2 and r_5 to C_0. These transition corresponds to $2 \rightarrow 0$ in the Markov model, and has then a probability



Last N transitions	Probability	
$r_5 : 2 \rightarrow 0$	$\mathbb{P}_1 = \mathbb{P}_{2\rightarrow 0}$	$= 1/5 = 0.2$
$r_6 : 0 \rightarrow 2$	$\mathbb{P}_2 = \mathbb{P}_1 \times 1/3$	$= 1/15 \approx 0.666$
$r_7 : 2 \rightarrow 2$	$\mathbb{P}_3 = \mathbb{P}_2 \times 3/5$	$= 1/25 = 0.04$
$r_8 : 2 \rightarrow 2$	$\mathbb{P}_4 = \mathbb{P}_3 \times 3/5$	$= 3/125 \approx 0.024$
$r_9 : 2 \rightarrow 1$	$\mathbb{P}_5 = \mathbb{P}_4 \times 1/5$	$= 1/3215 < T = 1/200$
	$\Rightarrow$ r_5 trigger an anomaly	

Figure 2: Trained Markov model and probability of the terminal path of length $N = 5$, with a threshold $T = 0.005$.

$\mathbb{P}_1 = 1/5$ to occur. Following the 5-step path from r_4 to r_9, this sequence has a probability of 1/3215 to happen, which is way below the anomaly threshold (set to 0.005 for this toy example).

2.2 Dataset

The molding machines of our dataset are equipped with a large array of sensors, measuring various parameters of their processing including distance, pressure, time, frequency, volume, temperature, time, speed, and force. The dataset is encoded as RDF [20] (Resource Description Framework) triples using Turtle [19] and consists of two types of inputs, namely a stream of measurements and a meta-data file. The stream measurements contain a sequence of *observation groups*, a 120 dimensional vector with the events from all sensors for a single time-tick and machine. It is noteworthy that the vector contains a mix of different value types, *e.g.,* text and numerical values. Each observation group is marked with a physical timestamp and has a machine identifier. In addition, each event contains a sensor identifier, a sensor reading and a sensor type. Each machine outputs a (complete) observation group once every second. W is the size in time of the sliding window and, in steady state, the exact count of the sliding window.

The query has to run against two different dataset types, namely static and dynamic. In the former, sensors from all machines listed in the meta-data output their events at a given rate. In the latter, machines can leave and join the *working set*. If a solution leverages data parallelism, by partitioning the input stream on machines and/or sensors, the machine's churn may become imbalanced.

2.3 Evaluation Platform

ACM DEBS 2017 Grand Challenge introduces a long awaited improvement over previous years, allowing the evaluation of the submitted solutions using a distributed environment. To provide a fair framework supporting the linked-data flavor of the challenge, the chosen evaluation platform is the automated evaluation platform provided by the European Unionfis H2020 HOBBIT [6] project.

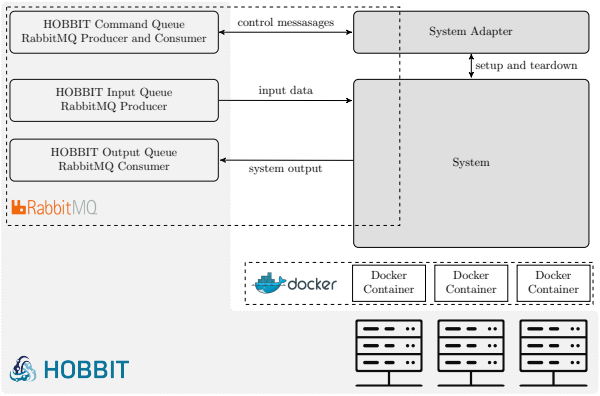


Figure 3: Evaluation Platform Architecture

HOBBIT aims at abolishing the barriers in the adoption and deployment of Big Linked Data by European companies, by means of open benchmarking reports that allow them to assess the fitness of existing solutions for their purposes. These benchmarks are based on data that reflects reality and measures industry-relevant Key Performance Indicators (KPIs) with comparable results using standardized hardware.

HOBBIT (Figure 3) enables running a system on a cluster of 3 physical servers equipped with a dual socket 64 bit Intel Xeon E5-2630v3 (8-Cores, 2.4 GHz, Hyperthreading, 20MB Cache) with 256 GB RAM and Gigabit Ethernet. The deployment is handled through Docker [2] containers, which package everything required to make a software run in an isolated environment. Unlike VMs, containers do not bundle a full operating system but only the libraries and settings required to make the software work. This makes for light-weight and self-contained systems, guaranteeing a *write once, run (almost) anywhere* property. The communication with the platform is (both data and control) is based on RabbitMQ [11] queues while an adapter handled the control messages from and to the platform.

3 SOLUTION ARCHITECTURE

In this section we outline the general architecture of our solution and its rationale. Figure 4 identifies the three main tasks of our system architecture, namely input, business logic and output. Considering the query and given the large amount of available memory ($3 \times 256GB$) and the limited amount of cpu (96 virtual cores), we chose to prioritize execution time and cpu usage over memory.

3.1 Input Task

The first task encodes the input data from the HOBBIT platform RabbitMQ Input Queue (Figure 3) and parses the incoming event (sensor readings) into the format expected by the Business Logic. Notice that while this may seem a trivial task, for low window sizes parsing turns out to be the most intensive task of the analytic pipeline and an incorrect interaction with the evaluation platform RabbitMQ queue may induce starvation and other drawbacks inherent to distributed and parallel computation. Since the observation group (*i.e.,* the input data unit) is encoded in RDF triples, the natural parsing approach is through an RDF-parsing library (*e.g.,* Apache Jena [16]), however the ease of use also comes with a large performance overhead. A straightforward alternative is to implement an ad-hoc RDF string parser, which does not improve much due to the high cost of string comparison and manipulation operations. Delving slightly deeper, one may use byte arrays as an underlying messages type. Our approach is indeed to directly parse the consumed byte array, thus minimizing the conversions from bytes and using fast byte comparison operation. We provide more details in Section 4.3. Each ingested observation group yields 55 events, thus this task has a large *count* selectivity of 55. On the other hand, the observation group is encoded in RDF triples with turtle, while the system events are encoded in a 5-tuple of basic types. In addition, only 55 of the 120 events grouped in the observation group are monitored, yielding a *space* selectivity pf 0.012. Finally, the RabbitMQ consumer (which has a low execution time) is in the same tasks of the parser to avoid cpu under-utilization.

3.2 Business Logic Task

This task implements the mechanics of the ACM DEBS 2017 Grand Challenge query. The initial description of the query naturally leads to the instantiation of a pipeline of three parallel tasks: clustering, Markov model training, and Markov model querying.

Each stage takes into account the current time-based sliding window with a count of exactly W events. We have that the clustering (using k-means) execution time lower and upper bounds are respectively $\Omega(k + W)$ and $O(M(W + k))$, where k is the number of clusters, W the window size, and M the maximum number of clustering iterations. Given the cluster assignment for each events in the sliding window the Markov model can be trained with $\Theta(W)$ time. Finally, the Markov model querying requires to replay the last N transitions in the Markov model to compute the probability of the resulting path, yielding $\Theta(N)$ and $O(W)$ time (by construction $N \leq W$). Using a monolithic approach, where the whole business logic is performed as a single task, the overall execution time lower and upper bounds are asymptotically[1] the same of the clustering tasks alone.

Each event may impact the current clustering, the Markov model training and querying, since each stage requires access to the whole result from the previous stage. Notice that here the output may be large, for instance the clustering has to output a cluster assignment for each event in the sliding window. Therefore, using a single task to run the business logic does not harm the overall asymptotically time complexity and avoids transferring (possibly through the network) large chunks of data.

3.3 Output Task

The final task serializes the anomalies and publish them to the HOBBIT platform RabbitMQ Output Queue (Figure 3). As will be discussed next, this task cannot be parallelized, and therefore $\gamma = 1$. Thus, assuming an infinite number of cores, the execution time of this task bounds the maximum throughput (or minimum latency) of the whole system. Optimize this task to avoid it becoming a bottleneck to the whole process, is a major aspect of our implementation.

[1] Abusing notation somewhat we have $\Omega(k + 3W)$ and $O(M(W + k) + 2W)$.

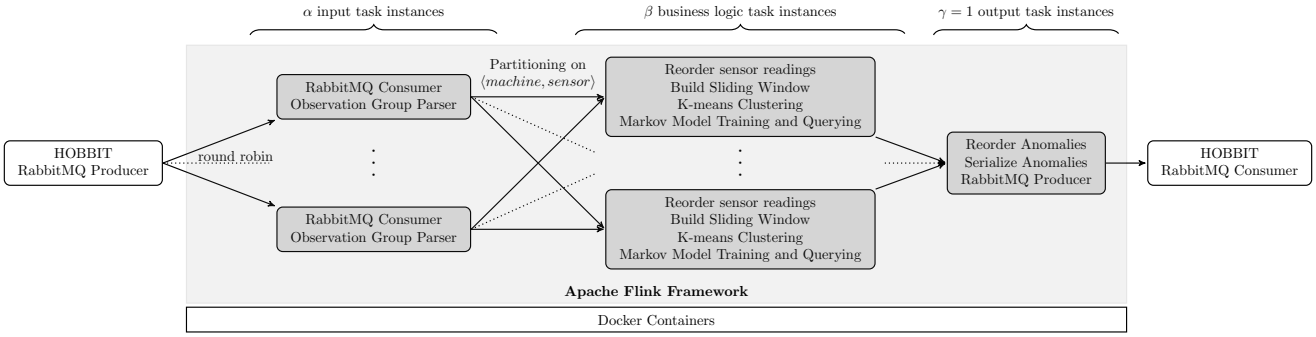


Figure 4: Solution Architecture

3.4 Parallelization and Distribution

The HOBBIT platform provides us with up to $8 * 2 * 3 = 48$ physical (or 96 hyper-threaded) cores, which utilization must be maximized to achieve good overall performances. This means that we spawn several instances of the aforementioned task. In particular we can spawn α instances (threads) running the input task which will consume messages from the HOBBIT platform RabbitMQ Input Queue and parse them in parallel. The HOBBIT platform uses the RabbitMQ Work Queues patterns, allowing only round-robing dispatching for multiple consumers.

Since the query is performed independently on each sensor of each machine, we can safely partition the events in β parts over the machine and sensor ids. We then spawn β instances for the business logic task, each receiving one of such parts, *i.e.,* the input to the business logic task from the input task is *key-grouped*. Notice that the parallelism in the input task may un-order the input of the business logic task: the n-th observation group for a given machine may finish its parsing before observation group $n + 1$. This compels us to introduce re-ordering step in the business logic task.

Considering the output task, the data-parallelism of the business logic task may, in its turn, un-order the anomaly output across sensors. This compels us to introduce re-ordering step in the output task. To avoid further re-ordering, the output task cannot be parallelized.

3.5 Apache Flink

The best performances are in general achieved by using ad-hoc underlying framework, and this has been the case for most previous edition of the ACM DEBS Grand Challenge. However, we strongly believe that using publicly available general purpose streaming engine is a more interesting choice for the DEBS community. We selected three initial candidates, Apache Storm [18], Apache Spark [17] and Apache Flink [15], and then further refined our selection considering the challenge query and our architecture requirements, as well as feature and performance comparisons [8, 14]. Finally we picked Apache Flink based on: **(i)** documented higher throughputs and lower latencies, **(ii)** API at high to low abstraction level, **(iii)** native time-based window and out-of-order managing mechanisms based on event-time, **(iv)** streamlined performance tuning, **(v)** both API and engine coded with HOBBIT's reference language (Java).

4 IMPLEMENTATION DETAILS

In this section we provide more details on our solution implementation, as well as the more relevant optimizations.

4.1 Load Balancing, Placement and Parallelism

Load balancing in distributed computing is a well known problem that has been extensively studied since the 80s. We can identify two ways to perform load balancing in stream processing systems [5]: either when placing the task instances on the available machines [1] or when assigning load to the parallelized instances of tasks [10, 12, 13]. In this setting we have complete control over both angles, since we known a-priori the values that drive the execution time of the tasks: W and k.

In particular we have (in the meta-data file) the k values for all sensors on all machines, which is in average per sensor equal to 50. We implemented a partitioning function, and an associated hash function, that partitions the sensors over the available β instances of the business logic task in order to hit the same per sensor average value of k. We leverage this mechanism also to attempt to minimize the impact of the machine's churn in the second scenario by spreading as much as possible the sensor of a single machine over the β instances.

With respect to placement, the pipeline architecture itself prevents avoidable hops over the network. However notice that, given the placement and the stream partitioning, a part may incur from 0 to 2 hops over the network. This variance is undesirable and a solution would be to place the tasks instances given the partitioning. An orthogonal concern is to avoid unbalancing the load on the the network interfaces and IP stacks of the available machines. To mitigate this issue we spread the α input task instances and the β business logic task instances evenly (on the node hosting Flink's JobManager process some cpu must be spared) among the 3 available machines.

Since we control the parallelism of both the input (α), the business logic (β) and output ($\gamma = 1$) tasks, we must strike the ratio that do not introduce a bottleneck. In other words we have to maximize β while satisfy the following two equations:

$$\alpha + \beta + \gamma + \mu \leq 3(32 - \sigma) \text{ and } \frac{1}{55}\overline{w}_\alpha \alpha \geq \overline{w}_\beta \beta$$

where μ is the number of cores allocated to Flink's JobManager process, σ is the number of cores allocated to the operating system

on each machine, while $\overline{w}_\alpha$ and $\overline{w}_\beta$ are respectively the average execution time of the input and business logic tasks collected through experimental evaluations. Notice that $\overline{w}_\beta$ is a function of k and W (*cf.,* Section 3.2). Finally, as mentioned in Section 3.3, we must also guarantee that the output task is not a bottleneck, *i.e.,* $\overline{w}_\beta \beta \geq \overline{w}_\gamma$.

4.2 Minimize Garbage Collection

We chose to develop our system not only to fit HOBBIT and Flinks API, but also for the dynamic optimizations [3] it performs while running, which typically benefit long running systems. In turns then out that the optimizations that a designer and/or programmer can introduce are mainly targeted at maximize resource usage while reducing contention, and keeping object creation and deletion under check. It is well known [9] that an excessive object allocation churn in the JVM may hit the overall performances by inducing too many garbage collector cycles. Aiming at minimizing object allocation (and deletion) at runtime, we maximized object re-usage. The most relevant instances are discussed in more details in the following Sections 4.4 and 4.5. Notice that this pattern is quite more error prone since it partially forfeit encapsulation. To mitigate this problem (**i**) we implemented this pattern only as the last optimization layer and (**ii**) we introduced proper interface to ease swapping from classical to object re-usage pattern with ease.

4.3 ByteArray Parser

As mentioned in Section 3.1, with low value of W the parsing becomes a bottleneck. To design a fast parser we wanted to avoid as much as possible string operations, type conversions and object creations. While the input data is a sequence of grouped events (observation groups) encoded in RDF triples with turtle (*i.e.,* string), the RabbitMQ messages underlying type is byte arrays. As such, we implemented an ad-hoc parser that works directly on the received byte arrays. Our implementation uses only displacement and access on the array as well as byte comparisons. Given the RDF ontology, we can identify a specific offset and character (byte) that uniquely identify the type of a triple. The parser main loop is the following: (**i**) the parser scans the byte array *bytes* from the current offset and identifies which type is the triple, (**ii**) then *bytes* and the current offset are passed to a type specific parser and (**iii**) the current offset is moved to the start of the following triple (*i.e.,* after the *new line* character) until the end of *bytes* is reached. For instance, if the current triple encodes the sensor identifier, we can compute the offsets (start and end) from the triple start position of the sensor identifier. The type specific parser extracts the identifier integer value using the method GETINTEGER (*cf.,* Listing 5). We introduced a number of other optimizations to reduce the number of accesses and comparisons (*i.e.,* fast skipping to the *new line* character).

4.4 Reordering events and anomalies

As mentioned in Section 3.4, parallelizing the input and business logic tasks requires to introduce two reordering stages. While we initially planned to leverage Flink's native mechanism to handle out-of-order events in sliding windows, we stumbled upon two shortcomings of the current implementations. The foremost is that while the engine waits for late events (*i.e.,* the window contains the correct events), it does not reorder the window (*i.e.,* the late event is wrongly positioned). The other issue with Flink's native out-of-order mechanism, is that it assumes that the first received timestamp is t_0 (*i.e.,* the application specific *origin* of time), while this may not be true in our setting. Since our application is strongly order sensitive, we had to implement an ad-hoc reordering mechanism that leverages the knowledge on the inter-arrival time of events (1 second). To identify the t_0 for each sensor we implement the following heuristic: (**i**) store all events in a buffer (*i.e.,* stall the execution) until there is a sequence of consecutive (*i.e.,* 1 second appart) events of length $\rho \times W$ (where ρ is a user defined parameter encoding the expected maximum lateness) rooted in the event with the lowest timestamp, (**ii**) use the root of the sequence as t_0. Once t_0 is chosen, reordering the output of the input task boils down to store in the same buffer early-arrivals and add to the window consecutive events.

It is then crucial that this buffer is backed by a highly performant data structure. LMAX Disruptor [7] are a well known technology which provides a strong performance boost by leveraging the concept of circular buffer which main goal is to minimize object allocation churn (*cf.,* Section 4.2). We took inspiration from this design and implemented our own circular buffer with $O(1)$ operations, called `pending` (Figure 6). `Pending` maintains a pointer (NEXT) to the entry associated with the next event. When polled, `pending` returns null if the next event has not arrived yet (*i.e.,* NEXT points an empty entry), otherwise it returns the next event and moves the NEXT forward. When a new event is added, `pending` computes in which entry it falls based on the distance between the new event timestamp and the timestamp of the event pointed by NEXT. The initial size of `pending` is set to $\rho \times W$ (where ρ is a user defined parameter encoding the expected maximum lateness). If an added event has to be added in an entry exceeding this size, `pending` automatically doubles it size. Notice that it in `pending` is easy to identify time gaps in among events, which may indicate that a machine has stopped sending events for a time interval.

```
1: function GETINTEGER(byte[] bytes, start, end, byte[] digits)
   ▷ digits is a static array storing the byte value of the 9 digits
2:     num ← 0
3:     for each i ∈ [0, end − start] and j ∈ [0, |digits| − 1] do
4:         if bytes[end − i] = digits[j] then
5:             num ← num + j × 10^i
6:     return num
```

Figure 5: Pseudo code of the byte to integer parser

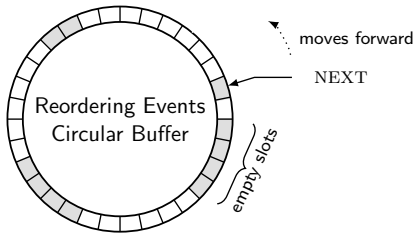


Figure 6: `Pending` buffer with $W = 16$ and $\rho = 2$

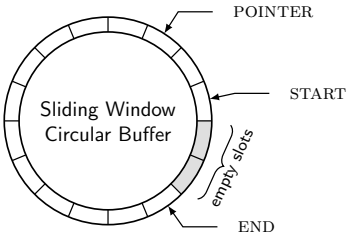


Figure 7: `Window` buffer with $W = 16$

Guaranteeing that anomalies are returned ordered, the output task must know if it has yet to receive any previous one. Toachieve that, the business logic does not filter out non-anomalous events, delegating the filtering stage to the output task. Then the output task receives all events (anomalous and non) and can reorder anomalies leveraging the logical timestamp associated to each observation group, the sensor identifier as well as the ordering guarantee among event of the same sensor. Similarly to the reordering mechanism of the business task, here we have to store events from the business logic that have arrived ahead of time. The data structure backing this mechanism is a slight variant of `pending`.

4.5 Sliding Window

Due to Flink's native windowing mechanism, the reordering stage could not be run inside the business logic task instances (*i.e.,* requiring an additional thread). To overcome this limitation and avoid wasting a thread, we had to implement our own time based sliding window mechanism. Also in this case the sliding window buffer must be backed by a highly performant data structure. We used again the concept of circular buffer, implementing another circular buffer with $O(1)$ operations, called `window` (Figure 7). `Window` maintains a pointer to the first event in the window (START) and to the last (END). `Window` offers an iterator interface (backed by POINTER) to scan, in order, the current window. When an event is added, `Window` places it in the entry following END. If it is the entry pointed by START, then the first item in the window is overwritten and START (*i.e.,* the window) moves forward. Notice that it in `window` is easy to retrieve the first, last and last but N event in the window.

4.6 RabbitMQ, Flink and JVM

In this section we discuss the configurations of the three main piece of software underlying our solutions, starting with the communication middleware.

Most of RabbitMQ configuration (*i.e.,* durability, auto-deletion, *etc.*) is bounded by the HOBBIT platform and there is not tuning possible for our RabbitMQ producer in the output task. Considering the input task, to ensure an exactly-once semantic in the message delivery, the HOBBIT data input queue producer requires an acknowledgment of the delivery. As is, this pattern can quickly choke the system throughput by introducing a round-trip time delay in between each message and forcing the producer to process an acknowledgment for each produced message. The former can me mitigated through the *consumer prefetch* configuration parameter which sets the number of messages the producer is allowed to send to a single consumer without acknowledgment. Enabling *multiple acknowledgments* the consumer can acknowledge several deliveries with a single acknowledgment message, effectively reducing the overhead.

Flink runs the submitted job task instances into `task manager` processes (JVMs), each handling a configurable amount of slots. Given n slots in a `task manager`, each has access to a n-th of the `task manager` available memory, can contain any number of threads and tasks instances (not belonging to the same task). The networking stack is shared among threads in the same `task manager` and thus may become a bottleneck. We then run 2 `task managers` per machine, each with a slot for each available physical core. Assigning a *resouce group* name to a task forces or avoids co-location of the instances of different tasks in a `task manager` slot. *Chaining* instead allows (or prevents) Flink to run two consecutive tasks instances in the same thread. These two mechanism configure the task instance-to-thread and the thread-to-slot allocations. It is also possible strike a good balance between throughput and latency configuring Flink's network level batching, *i.e.,* setting the batching timeout from 0 to ∞. We set Flink's parameter `taskmanager.memory.fraction` to 0 since our application does not benefit from Flink's internal memory management.

Finally, the JVMs of the 2 `task managers` are set to use almost all the available RAM (256 GB), leaving a generous slack to the operating system.

REFERENCES

[1] V. Cardellini, V. Grassi, F. Lo Presti, and M. Nardelli. Optimal Operator Placement for Distributed Stream Processing Applications. In *Proc. of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS, 2016.
[2] Docker, Inc. Docker. https://www.docker.com/.
[3] B. Goetz. Dynamic compilation and performance measurement. https://www.ibm.com/developerworks/library/j-jtp12214/. IBM.
[4] V. Gulisano, Z. Jerzak, R. Katerinenko, M. Strohbach, and H. Ziekow. The DEBS 2017 grand challenge. In *Proc. of the 11th ACM International Conference on Distributed and Event-based Systems, Barcelona, Spain,*, DEBS, 2017.
[5] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A Catalog of Stream Processing Optimizations. *ACM Computing Surveys*, 46(4):41–34, 2014.
[6] HOBBIT. Holistic Benchmarking of Big Linked Data. https://project-hobbit.eu/.
[7] LMAX-Exchange. LMAX Disruptor. https://lmax-exchange.github.io/disruptor/.
[8] Lopez, J. and Vieru, M. Apache Showdown: Flink vs. Spark. https://tech.zalando.com/blog/apache-showdown-flink-vs.-spark/?gh_src=4n3gxh1. Zalando.
[9] N. Nagarajayya and J. S. Mayer. Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory. http://www.oracle.com/technetwork/systems/index-156457.html. Oracle.
[10] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, and M. Serafini. When two choices are not enough: Balancing at scale in distributed stream processing. In *Proc. of the IEEE 32nd International Conference on Data Engineering*, ICDE, 2016.
[11] Pivotal Software, Inc. RabbitMQ. https://www.rabbitmq.com/.
[12] N. Rivetti, E. Anceaume, Y. Busnel, L. Querzoni, and B. Sericola. Proactive Online Scheduling for Shuffle Grouping in Distributed Stream Processing Systems. In *Proc. of the 17th ACM/IFIP/USENIX International Middleware Conference*, 2016.
[13] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola. Efficient Key Grouping for Near-Optimal Load Balancing in Stream Processing Systems. In *Proc. of the 9th ACM Intl Conf. on Distributed Event-Based Systems*, DEBS, 2015.
[14] N. Spangenberg, M. Roth, and B. Franczyk. Evaluating new approaches of big data analytics frameworks. In *Proc. of the 18th International Conference in Business Information Systems, Poznań, Poland*, BIS, 2015.
[15] The Apache Software Foundation. Apache Flink. https://flink.apache.org/.
[16] The Apache Software Foundation. Apache Jena. http://jena.apache.org/.
[17] The Apache Software Foundation. Apache Spark. http://spark.apache.org/.
[18] The Apache Software Foundation. Apache Storm. http://storm.apache.org.
[19] World Wide Web Consortium. W3C Turtle. https://www.w3.org/TR/turtle/.
[20] World Wide Web Consortium (W3C). W3C Resource Description Framework (RDF). https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.