Online trajectory analysis with scalable event recognition

Emmanouil Ntoulias
NCSR "Demokritos"
manosntoulias@iit.demokritos.gr

Elias Alevizos
National and Kapodistrian University of Athens
NCSR "Demokritos"
ilalev@di.uoa.gr

Alexander Artikis
University of Piraeus
NCSR "Demokritos"
a.artikis@unipi.gr

Athanasios Koumparos
Vodafone Innovus
athanasios.koumparos@vodafoneinnovus.com

ABSTRACT

Moving object monitoring is becoming essential for companies and organizations that need to manage thousands or even millions of commercial vehicles or vessels, detect dangerous situations (e.g., collisions or malfunctions) and optimize their behavior. It is a task that must be executed in real-time, reporting any such situations or opportunities as soon as they appear. Given the growing sizes of fleets worldwide, a monitoring system must be highly efficient and scalable. It is becoming an increasingly common requirement that such monitoring systems should be able to automatically detect complex situations, possibly involving multiple moving objects and requiring extensive background knowledge. Building a monitoring system that is both expressive and scalable is a significant challenge. Typically, the more expressive a system is, the less flexible it becomes in terms of its parallelization potential. We present a system that strikes a balance between expressiveness and scalability. Our proposed system employs a formalism that allows analysts to define complex patterns in a user-friendly manner while maintaining unambiguous semantics and avoiding ad hoc constructs. At the same time, depending on the problem at hand, it can employ different parallelization strategies in order to address the issue of scalability. Our experimental results show that our system can detect complex patterns over moving entities with minimal latency, even when the load on our system surpasses what is to be realistically expected in real-world scenarios.

1 INTRODUCTION

Commercial vehicle fleets constitute a major part of Europe's economy. There were approximately 37 million commercial vehicles in the European Union in 2015[1] and this number is growing every year with an increasing rate. Devices emitting spatial and operational information are installed on commercial vehicles. This information helps fleet management applications improve the management and planning of transportation services [31].

Consider another case of monitoring of moving objects, equally important from an economic and environmental point of view: maritime monitoring systems. Such systems have been attracting considerable attention for economic as well as environmental reasons [24, 29, 30]. The Automatic Identification System (AIS) [2] is used to track vessels at sea in real-time through data exchange with other ships nearby, coastal stations, or even satellites. Cargo ships of at least 300 gross tonnage and all passenger ships, regardless of size, are nowadays required to have AIS equipment installed and regularly emit AIS messages while sailing at sea. Currently, there are more than 500.000 vessels worldwide that can be tracked using AIS technology[3]. It is crucial, both for authorities and for maritime companies, to be able to track the behavior of ships at sea in order to avoid accidents and ensure that ships adhere to international regulations.

Streams of transient data emitted from vehicles or ships must be processed with minimal latency, if a monitoring system is to provide significant margins for action in case of critical situations. We therefore need to detect complex patterns of interest upon these streams in an online and highly efficient manner that can gracefully scale as the number of monitored entities increases. Besides kinematic data, it is also important to be able to take into account static (or "almost" static, with respect to the rate of the streaming data), background knowledge, such as weather data, point of interest (POI) information (like gas stations, ports, parking lots, police departments, etc. [21], NATURA areas where ships are not allowed to sail, etc. This enhanced data stream produces valuable opportunities for the detection of complex events. One can identify certain routes a vehicle or a ship is taking, malfunctions in the device installed, in the GPS tracker or the AIS transponder, cases of illegal shipping in protected areas or possible collisions between ships moving dangerously close to each other, to name but a few of the possible patterns which could be of interest to analysts.

As a solution to the problem of monitoring of moving objects, we present a Complex Event Processing system that aims to improve the operating efficiency of a commercial fleet. It operates online with enriched data in a streaming environment. Our contributions are the following:

- We present a Complex Event Processing (CEP) system based on symbolic automata which allows analysts to define complex patterns in a user-friendly language. Our proposed language has formal semantics, while being able to also take into account background knowledge.
- We define a series of realistic complex patterns that identify routes and malfunctions of vehicles and detect critical situations for vessels at sea.
- We present and compare various implementations of parallel processing techniques and discuss their applicability.
- We test our approach using large, real-world, heterogeneous data streams from diverse application domains, showing that we can achieve real-time performance even

---

[1]http://www.acea.be/statistics/article/vehicles-in-use-europe-2017
[2]http://www.imo.org/OurWork/Safety/Navigation/Pages/AIS.aspx

[3]https://www.vesselfinder.com

in cases of significantly increased load, beyond the current demand levels.

The remainder of this paper is organized as follows. Section 2 discusses related work, while Section 3 describes our CEP engine. In Section 4 the distributed version of our engine is presented. Section 5 summarizes the datasets of vehicle and vessel traces and defines the recognition patterns. It also presents our empirical evaluation. Finally, we conclude the paper in section 6 and describe our future work.

2 RELATED WORK

Complex event recognition systems accept as input a stream of time-stamped, "simple, derived events" (SDEs). These SDEs are the result of applying a simple transformation to some other event (e.g., a measurement from a sensor). By processing them, a CEP engine can recognize complex events (CEs), i.e. collections of SDEs satisfying some pattern. There are multiple CEP systems proposed in the literature during the last 15 years, falling under various classes [16, 19]. Automata-based systems constitute the most common category. They compile patterns (definitions of complex events) into finite state automata, which are then used to consume streams of simple events and report matches whenever an automaton reaches a final state. Examples of such systems may be found in [5, 11, 18, 28, 32]. Another important class of CEP systems are the logic-based ones. In this case, patterns are defined as rules, with a head and a body defining the conditions which, if satisfied, lead to the detection of a CE. A typical example of a logic-based system may be found in [12]. Finally, there are some tree-based systems, such as [22, 23], which are attractive because they are amenable to various optimization techniques.

For efficient processing on big data streams, distributed architectures need to be employed [19]. Big data platforms, such as Apache Spark and Storm, have been used to embed CEP engines into their operators. Both platforms have incorporated Sidhi [7, 9] and Esper [3, 4] as their embedded engines. Flink [1], on the other hand, provides support for CEP with the FlinkCEP built-in library [5]. Besides using these Big Data platforms, numerous other parallelization techniques have been proposed in the literature that can achieve a more fine-grained control over how the processing load is distributed among workers. Pattern-based parallelization is the most obvious solution, where the patterns are distributed among the processing units [15]. One disadvantage of this parallelization scheme is that events have to be replicated to multiple processing units, since a new input event may need to be consumed by more than one pattern. Moreover, the parallelization level is necessarily limited by the number of patterns (for a single pattern, this method offers no benefits). Operator-based parallelization constitutes another approach, where the CEP operators are assigned to different processing units [13, 28]. This allows for multi-pattern optimizations and avoids the data replication issue of the previous technique. On the other hand, the parallelization level is again limited, this time by the number of operators present in the pattern (which is closely related to the number of automaton states in automata-based CEP systems). Finally, in data-parallelization schemes, events are split among multiple instances of the same pattern [20]. For example, a pattern trying to detect violations of speed limits must be applied to all the monitored vehicles and thus the input stream may be partitioned according to the id of the vehicles. The advantage of this method is that it can scale well with the input event rate. It is, however, not always obvious how an input stream should be partitioned, while avoiding data replication.

3 AUTOMATA-BASED EVENT RECOGNITION

We begin by first presenting our framework for CEP. It is based on Wayeb, a Complex Event Processing and Forecasting engine which employs symbolic automata as its computational model [10, 11]. The rationale behind our choice of Wayeb is that, contrary to other automata-based CEP engines, it has clear, compositional semantics due to the fact that symbolic automata have nice closure properties [17]. At the same time, it is expressive enough to support most of the common CEP operators [19], while remaining amenable to the standard parallelization solutions. In this paper, we extend Wayeb's language in order to support more expressive patterns.

Symbolic automata constitute a variation of classical automata, with the main difference being that their transitions, instead of being labeled with a symbol from an alphabet, are equipped with formulas from Boolean algebra [17]. A symbolic automaton consumes strings and, after every new element, applies the predicates of its current state's outgoing transitions to that element. If a predicate evaluates to TRUE then the corresponding transition is triggered and the automaton moves to that transition's target state. A Boolean algebra is defined as follows:

Definition 3.1 (Effective Boolean algebra [17]). A Boolean algebra is a tuple $(\mathcal{D}, \Psi, \llbracket\_\rrbracket, \bot, \top, \vee, \wedge, \neg)$ where $\mathcal{D}$ is a set of domain elements; Ψ is a set of predicates closed under the Boolean connectives; $\bot, \top \in \Psi$; the component $\llbracket\_\rrbracket : \Psi \to 2^{\mathcal{D}}$ is a denotation function such that $\llbracket\bot\rrbracket = \emptyset$, $\llbracket\top\rrbracket = \mathcal{D}$ and $\forall \phi, \psi \in \Psi$: a) $\llbracket\phi \vee \psi\rrbracket = \llbracket\phi\rrbracket \cup \llbracket\psi\rrbracket$; b) $\llbracket\phi \wedge \psi\rrbracket = \llbracket\phi\rrbracket \cap \llbracket\psi\rrbracket$; and c) $\llbracket\neg\phi\rrbracket = \mathcal{D} \setminus \llbracket\phi\rrbracket$.

Elements of $\mathcal{D}$ are called *characters* and finite sequences of characters are called *strings*. A set of strings $\mathcal{L}$ constructed from elements of $\mathcal{D}$ ($\mathcal{L} \subseteq \mathcal{D}^*$, where * denotes Kleene-star) is called a language over $\mathcal{D}$.

Wayeb uses symbolic regular expressions to define patterns and to represent a class of languages over $\mathcal{D}$. Wayeb's standard operators are those of the classical regular expressions, i.e., concatenation, disjunction and Kleene-star. We extend Wayeb to include various extra CEP operators: that of negation and those of different selection policies (see [19] for a discussion of selection policies). Symbolic regular expressions are defined as follows:

Definition 3.2 (Symbolic regular expression). A Wayeb symbolic regular expression (*SRE*) over a Boolean algebra $(\mathcal{D}, \Psi, \llbracket\_\rrbracket, \bot, \top, \vee, \wedge, \neg)$ is recursively defined as follows:

- If $\psi \in \Psi$, then $R := \psi$ is a symbolic regular expression, with $\mathcal{L}(\psi) = \llbracket\psi\rrbracket$, i.e., the language of ψ is the subset of $\mathcal{D}$ for which ψ evaluates to TRUE;
- Disjunction / Union: If R_1 and R_2 are symbolic regular expressions, then $R := R_1 + R_2$ is also a symbolic regular expression, with $\mathcal{L}(R) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$;
- Concatenation / Sequence: If R_1 and R_2 are symbolic regular expressions, then $R := R_1 \cdot R_2$ is also a symbolic regular expression, with $\mathcal{L}(R) = \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)$, where $\cdot$ denotes concatenation. $\mathcal{L}(R)$ is then the set of all strings constructed from concatenating each element of $\mathcal{L}(R_1)$ with each element of $\mathcal{L}(R_2)$;
- Iteration / Kleene-star: If R is a symbolic regular expression, then $R' := R^*$ is a symbolic regular expression, with

$\mathcal{L}(R^*) = (\mathcal{L}(R))^*$, where $\mathcal{L}^* = \bigcup_{i \geq 0} \mathcal{L}^i$ and $\mathcal{L}^i$ is the concatenation of $\mathcal{L}$ with itself i times.

- Bounded iteration: If R is a symbolic regular expression, then $R' := R^{x+}$ is a symbolic regular expression, with $R^{x+} = \overset{x \ times}{R \cdot \cdots \cdot R} \cdot R^*$.

- Negation / complement: If R is a symbolic regular expression, then $R' := !R$ is a symbolic regular expression, with $\mathcal{L}(R') = (\mathcal{L}(R))^c$.

- skip-till-any-match selection policy: If $R_1, R_2, \cdots, R_n$ are symbolic regular expressions, then $R' := \#(R_1, R_2, \cdots, R_n)$ is a symbolic regular expression, with $R' := R_1 \cdot \top^* \cdot R_2 \cdot \top^* \cdots \top^* \cdot R_n$.

- skip-till-next-match selection policy: If $R_1, R_2, \cdots, R_n$ are symbolic regular expressions, then $R' := @(R_1, R_2, \cdots, R_n)$ is a symbolic regular expression, with $R' := R_1 \cdot !(\top^* \cdot R_2 \cdot \top^*) \cdot R_2 \cdots !(\top^* \cdot R_n \cdot \top^*) \cdot R_n$.

A Wayeb expression without a selection policy implicitly follows the strict-contiguity policy, i.e., the SDEs involved in a match of a pattern should occur contiguously in the input stream. The other two selection policies relax the strict requirement of contiguity (see [19] for details). Note that all these operators, even those of selection policies, may be arbitrarily used and nested in an expression, without any limitations. This is in contrast to other CEP systems where nested operations may be prohibited.

Wayeb patterns are defined as symbolic regular expressions which are subsequently compiled into symbolic automata. The definition for a symbolic automaton is the following:

Definition 3.3 (Symbolic finite automaton [17]). A symbolic finite automaton (*SFA*) is a tuple $M = (\mathcal{A}, Q, q^s, F, \Delta)$, where $\mathcal{A}$ is an effective Boolean algebra; Q is a finite set of states; $q^s \in Q$ is the initial state; $Q^f \subseteq Q$ is the set of final states; $\Delta \subseteq Q \times \Psi_\mathcal{A} \times Q$ is a finite set of transitions.

A string $w = a_1 a_2 \cdots a_k$ is accepted by a *SFA M* iff, for $1 \leq i \leq k$, there exist transitions $q_{i-1} \overset{a_i}{\rightarrow} q_i$ such that $q_0 = q^s$ and $q_k \in Q^f$. The set of strings accepted by M is the language of M, denoted by $\mathcal{L}(M)$. It can be proven that every symbolic regular expression can be translated to an equivalent (i.e., with the same language) symbolic automaton [17].

We are now in a position to precisely define the meaning of "complex events". Input events come in the form of tuples with both numerical and categorical values. These tuples constitute the set of domain elements $\mathcal{D}$. A stream S is an infinite sequence $S = t_1, t_2, \cdots$, where each t_i is a tuple ($t_i \in \mathcal{D}$). Our goal is to report the indices i at which a CE is detected. If $S_{1..k} = \cdots, t_{k-1}, t_k$ is the prefix of S up to the index k, we say that an instance of a *SRE* R is detected at k iff there exists a suffix $S_{m..k}$ of $S_{1..k}$ such that $S_{m..k} \in \mathcal{L}(R)$. If we attempted to detect CEs, as defined above, by directly compiling an expression R to an automaton, we would fail. Consider, for example, the (classical) regular expression $R := a \cdot b$ and the (classical) stream/string $S = a, b, c, a, b, c$. If we compile R to a (classical) automaton and feed S to it, then the automaton would reach its final state after reading the second element t_2 of the string. However, it would then never reach its final state again. We would like our automaton to reach its final state every time it encounters a, b as a suffix, e.g., again after reading t_5 of S. We can achieve this with a simple trick. Instead of using R, we first convert it to $R_s = \top^* \cdot R$. Using R_s we can detect CEs of R while consuming a stream S, since a stream segment $S_{m..k}$ is recognized by R iff the prefix $S_{1..k}$ is recognized by R_s.

Table 1: An example stream composed of six events. Each event has a vehicle identifier, a value for that vehicle's speed and a timestamp.

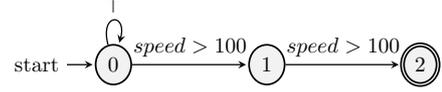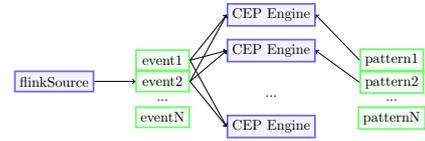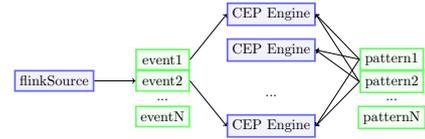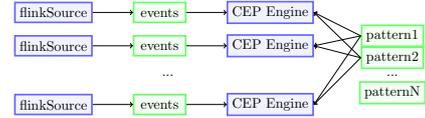| vehicle id	78986	78986	78986	78986	78986	...
speed	85	93	99	104	111	...
timestamp	1	2	3	4	5	...

Figure 1: Streaming symbolic automaton created from the expression $R := (speed > 100) \cdot (speed > 100)$.

(a) Pattern-based parallelization.

(b) Partition-based parallelization.

(c) Special case of Partition-Based parallelization when one-to-one relation between sources and CEP engines exists.

Figure 2: Parallel schemes used with Wayeb.

The prefix $\top^*$ lets us skip any number of events from the stream and start recognition at any index $m, 1 \leq m \leq k$.

As an example, consider the domain of vehicle monitoring. An analyst could use the Wayeb language to define the pattern $R := (speed > 100) \cdot (speed > 100)$ in order to detect speed violations on roads where the maximum allowed speed is 100 km/h. This pattern detects two consecutive events where the speed exceeds the threshold in order to avoid cases where a vehicle momentarily exceeds the threshold, possibly due to some measurement error. This pattern would be compiled to the (non-deterministic) automaton of Figure 1. Table 1 shows an example stream processed by this automaton. For the first three input events, the automaton would remain in its start state, state 0. After the fourth event, it would move to state 1 and after the fifth event it would reach its final state, state 2. We would thus say that a complex event R was detected at *timestamp* = 5.

4 SCALABLE EVENT RECOGNITION OVER MULTIPLE TRAJECTORIES

We now discuss how various parallelization schemes may be applied to our CEP engine. For this purpose, we leverage a popular

streaming platform, Apache Flink [1, 14]. Flink is a distributed processing engine for stateful computations over unbounded and bounded data streams. It is designed to run in cluster environments and perform computations at in-memory speed. In this paper, we focus on two parallelization techniques: pattern-based and partition-based parallelization [19]. We currently exclude state-based parallelization, since, as explained above, its parallelization level is limited by the number of automaton states, which is typically quite low (it is often a single-digit number). We do intend, however, to examine in the future how it could be combined with pattern- or partition-based parallelization to provide them with an extra performance boost.

In pattern-based parallelization, each available CEP engine receives a unique subset of the patterns and performs recognition for these patterns only, with these subsets being (almost) equal in size (see Figure 2a). On the other hand, the stream is broadcast to all parallel instances of any downstream operators. This is a significant (yet unavoidable) drawback of pattern-based parallelization, since each worker has a subset of the patterns while each pattern may need to process the whole stream. Note that each blue rectangle in Figure 2a represents a single thread. This means that in this architecture we have one thread for the source plus as many threads as the parallelism of the CEP operator.

In partition-based parallelization the opposite happens. Every CEP engine is initialized with all patterns, but the stream is not broadcast (see Figure 2b). A partitioning function is used to decide where each new input event should be forwarded. This function takes as input any attribute of the event (we use the id of a vehicle or vessel) and, by performing hashing, it outputs which parallel instance of the next operator the event will go to. As with pattern-based parallelization, we have one thread for the source plus as many threads as the parallelism of the CEP operator.

Besides Flink, we also use the Apache Kafka messaging platform to connect our stream sources to Wayeb instances [2]. Kafka provides various ways to consume streams. So far, we have focused on linear streams, i.e., events are assumed to be totally ordered and arrive at our system sequentially one after another. With Kafka, however, there is the option of using parallel input streams. A Kafka input topic can have multiple partitions and each partition can be consumed in parallel by a different consumer. In this case the input stream is already partitioned on some attribute of the events.

Through this Kafka functionality, a variant of partition-based parallelization becomes possible, where both the input source and the recognition engine work in parallel (see Figure 2c). If the parallelism of the input source (i.e., the number of partitions of the topic is the same as that of the recognition operator (i.e., number of CEP engines), then we can simply attach each source instance to a CEP engine instance without further re-partitioning on our end. When, however, the parallelisms are different, further re-partitioning is performed by partitioning each source the same way we partitioned the single threaded source. Unlike the previous architectures, each pair of a source and a CEP operator parallel instances belong in a single thread. This is attributed to operator chaining [8], a Flink mechanism that chains operators of the same parallelism in a single thread for better performance.

Similarly to partition, we can have multiple sources for pattern based parallelism as well. Messages in Kafka are still partitioned by a desired attribute, albeit we always have to perform the broadcasting step for each source. Hence, we don't have a variant of pattern parallelization, rather we use it only as a way to process streams of greater input rate.

5 EXPERIMENTAL EVALUATION

We present an extensive experimental evaluation of our parallel CEP engine on two datasets containing *real-world* trajectories of moving objects. The first dataset comes from the domain of fleet management for vehicles moving on roads and emitting information about their status. The second dataset consists of vessel trajectories from ships sailing at sea. In both cases, our goal is to simultaneously monitor thousands of moving objects and detect interesting (or even critical) behavioral patterns in real-time, as defined by domain experts. All experiments were conducted on a server with 24 processors. Each processor is an Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz. The server has 252 GB of RAM. The source code for Wayeb may be found in the following repository: https://github.com/ElAlev/Wayeb.

5.1 Fleet Management

Efficient fleet management is essential for transportation and logistics companies. We show how our proposed solution can effectively help in this task. With the help of experts, we define a set of patterns to be detected on real-time streams of trajectories and show that our engine can detect these patterns with an efficiency that is orders of magnitude better than real-time.

5.1.1 Dataset Description. The dataset is provided by Vodafone Innovus[4], our partner in the Track & Know project, which offers fleet management services. It contains approximately 270M records (243GB). It covers a period of 5 months, from June 30, 2018 11:00:00 PM to November 30, 2018 11:59:59 PM. The initial source emitting the data is composed of GPS (Global Positioning System) traces of moving vehicles. The data also includes speed information provided by an installed accelerometer and information regarding the level of fuel in a vehicle's tank measured by a fuel sensor. It is also enriched with weather and point-of-interest (POI) information (e.g., if a vehicle is close to a gas station, a university, a school etc), as described in [31]. Duration, accelaration and distance are some extra attributes that are calculated on the fly as they enter our system by storing information from previous events.

5.1.2 Pattern Definitions. The first pattern we have defined concerns vehicle routes. A route is the basic element of vehicle management and aggregates data between the start and the end point of a vehicle's motion cycle. A motion cycle is based on the engine status. Each vehicle route must start and end with an "engine-off" message, i.e., a message whose engine status attribute is "off". According to Vodafone Innovus, there are 12 patterns that describe the most frequent routes. These 12 route patterns can be expressed with a single Wayeb pattern as follows:

Definition 5.1. A route pattern for a vehicle is defined as the following sequence: emitting "engine-off" messages for at least 30 minutes, emitting at least one "moving" message and again emitting "engine-off" messages for at least 30 minutes:

$$Route := (Engine = Off \wedge Duration > 30) \cdot$$
$$(Engine = Moving)^{+} \cdot$$
$$(Engine = Off \wedge Duration > 30)$$

Unfortunately, the expected data flow can be corrupted due to a variety of reasons. These reasons include bad connection during the device installation or after the vehicle has been serviced, movement of the satellites, hardware malfunctions or, simply,

---

[4]https://www.vodafoneinnovus.com/

just an issue with the GPS. The result of these reasons is reflected in the data. For example, coordinates may change even though the vehicle is not moving or the vehicle may be moving but the coordinates remain the same. It is also often the case that the engine status is incorrect (e.g., parked messages are emitted even though engine is on, vehicle is moving yet engine status is not moving etc). These issues are important and need to be detected. We have summarized those issues in a number of patterns, defined as follows:

Definition 5.2. ParkedMovingSwing. Engine status swings between "parked" and "moving" during consecutive events.

$$ParkedMovingSwing := (Engine = Parked) \cdot (Engine = Moving) \cdot$$
$$(Engine = Parked) \cdot (Engine = Moving)$$

Definition 5.3. IdleParkedSwing. Engine status swings between "idle" and "parked" during consecutive events.

$$IdleParkedSwing := (Engine = Idle) \cdot (Engine = Parked) \cdot$$
$$(Engine = Idle) \cdot (Engine = Parked)$$

Definition 5.4. SpeedSwing. Speed swings between 0km/h and greater than 50km/h during consecutive events.

$$SpeedSwing := (Speed > 50) \cdot (Speed = 0) \cdot (Speed > 50) \cdot (Speed = 0)$$

Definition 5.5. MovingWithZeroSpeed. Engine status is "moving", distance traveled is greater than 30m, yet speed is 0km/h for more than 3 consecutive messages.

$$MWZS := (Engine = Moving \ \wedge \ Speed = 0 \ \wedge \ Distance > 30)^{3+}$$

Definition 5.6. MovingWithBadSignal. Vehicle is accelerating and distance traveled is greater than 30m yet there are no satellites tracking the vehicle for more than 3 consecutive messages.

$$MWBS := (Accelaration \ \wedge \ Satellites = 0 \ \wedge \ Distance > 30)^{3+}$$

In addition to the above issues, possibly related to malfunctions, experts are also interested in the following patterns:

Definition 5.7. Possible Theft. Engine status is parked, speed is 0km/h and distance traveled is greater than 30m for more than 3 consecutive messages.

$$PossibleTheft := (Engine = Parked \ \wedge$$
$$Speed = 0 \ \wedge \ Distance > 30)^{3+}$$

Definition 5.8. Dangerous Driving. There is ice on the road and the vehicle is moving above a specific speed limit for at least 2 consecutive messages.

$$DangerousDriving := (IceExists = TRUE \ \wedge \ Speed > v_{limit})^{2+}$$

Definition 5.9. Refuel Opportunity. Vehicle is close to a gas station and the fuel in the tank is less than 50% for at least 2 consecutive messages.

$$RefuelOpp := (CloseToGasStation = TRUE \ \wedge \ FuelLevel < 0.5)^{2+}$$
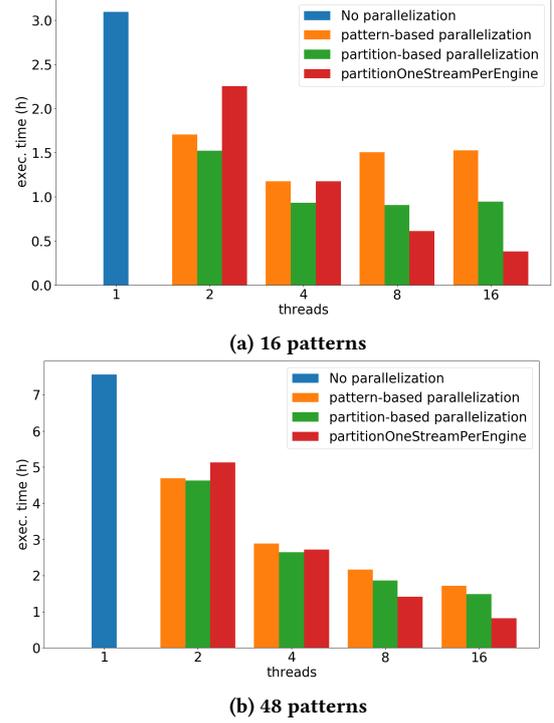

(a) 16 patterns

(b) 48 patterns

Figure 3: Recognition times of different parallelization techniques for different workloads. The horizontal axis represents the number of worker threads.

5.1.3 Recognition Results. Figure 3a showcases recognition times for our various parallelization techniques: pattern-based, partition-based and partition-based with one source per engine. We also show results for the non-parallel version. We have duplicated some of the patterns defined previously to simulate a greater workload of 16 patterns. We have repeated the experiment for 1, 2, 4 ,8 and 16 cores. Compared to the original single-core version, all three parallelization techniques exhibit speed-ups. For partition- and pattern-based parallelization, however, there seems to be an upper limit on the number of cores it is most efficient to use. For pattern-based parallelization there is a significant raise in time after 4 cores, while for partition-based there is no improvement after 2 cores. The reason is that the single source acts as a bottleneck. For partition-based parallelization we have one thread (the partitioner) deciding in which core each event will be forwarded, while for pattern-based events are broadcast to all cores, again in a single threaded manner. This explanation is supported by the smooth decrease in time when a parallel source is used for partition-based parallelization. While it starts off worse than pattern- and partition-based, it exhibits the best results for 16 cores.

To further support our claim above, we conducted a second experiment with a larger load, as shown in Figure 3b. 48 patterns were used this time (replicated in similar fashion as before) without any other changes. Indeed, with every recognition node having more work to do, execution time becomes less dependent on partitioning/broadcasting and more dependent on the actual recognition. Hence, speed-ups are now visible even for higher number of cores. As suspected, for parallel sources the results are not affected. Partition-based parallelization with parallel sources is slower when few threads are used (e.g., for 2 threads). This is
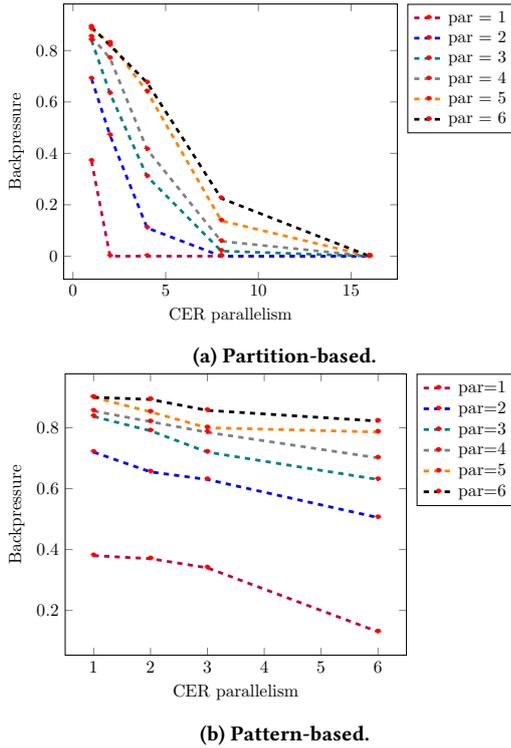
(a) Partition-based.

(b) Pattern-based.

Figure 4: Backpressure experiments for fleet management. The horizontial axis expresses the number of recognition threads. Workload is 6 patterns. Each dashed line represents a different parallelism of the source stream.

because the other two techniques have an extra thread handling the whole stream source and the work is in fact split between this one source thread and two others performing recognition. We thus have 3 threads performing similar volumes of work. When parallel sources are used, however, each parallel source is chained to a Wayeb engine in a single thread and we thus have 2 threads doing more work. Each thread handles half the source and performs recognition on half the stream.

In order to evaluate recognition speed independently from source speed we had to turn operator chaining off as we can't measure them separately when they belong in the same thread (i.e., in the case of one source per CEP engine). In addition, we leveraged parallel sources to achieve input streams of higher input rate. The goal here is to determine if our system can process input events faster than the source produces them. Flink offers a metric for this purpose, called *backpressure* [6]. Backpressure is judged on the availability of output buffers. Assuming that some task A sends events to some task B, if there is no output buffer available for task A, we say that task B is back pressuring task A. In our case A, is the source operator and B is the operator with the CEP Engine. 100 samples (each sample checks if there is any output buffer available) are triggered every 50ms in order to measure backpressure. The resulting ratio notifies us how many of these samples were indicating back pressure, e.g. 0.6 indicates that 60 in 100 were stuck requesting buffers from the network stack. According to the documentation [6] a ratio between 0 and 0.1 is normal. 0.1 to 0.5 is considered to be low and anything above 0.5 is high. Note that low and high pressure will slow down the source to match the throughput of the pressuring operator.

Results for Partition-based distribution are presented in Figure 4a. The backpressure ratio is plotted against the number of threads used for recognition. Generally, the more threads used the faster Wayeb can process events and hence less pressure is noted. Each dashed line represents a source with parallelism varying from 1 to 6. The event rate of a parallel source is measured by executing an experiment with 0% backpressure (i.e., it will not slow down due to pressure) and summing the event rate of each parallel instance presented by the flink dashboard (e.g., for 2 parallel instances of 120K events/second (e/s) for each one the overall sum is $120 + 120 = 240K$ (e/s)). Although, the greater the parallelism of the source the larger the event rate, it is not a multiplier as for 1, 2, 3, 4, 5 and 6 sources the rate becomes 130K, 240K, 350K, 430K, 440K and 490K e/s respectively. The workload in this experiment tries to emulate a real scenario and hence the route and the 5 malfunction patterns are used (i.e., they are not replicated). The results show that Wayeb can effectively process streams of at least 490K e/s (black line) for this workload as 0 pressure is exhibited when 16 workers are used. As it was stated before, the duration of the dataset is 5 months which translates to roughly 13M seconds. Since the total number of input events is 270M, the average event input rate is $270M/13M \approx 20\ e/s$. Comparing the pattern throughput rate (490K e/s) with the event input rate clearly exhibits a performance that is 4 orders of magnitude better than the real-time requirements of this use case.

We perform a similar experiment for pattern-based parallelization. This time the number of threads used for recognition varies between 1, 2, 3 and 6 as there are only 6 patterns - a handicap of this technique discussed earlier. Figure 4b showcases the results of our experiment. Unfortunately, even with 6 recognition threads and a single threaded source (purple dashed line) there is about 13% backpressure. Due to this, all sources are being slowed down to 110K e/s regardless of their parallelism. There is a drop in pressure the more recognition threads are used due to the better distribution of the patterns. However, it is not significant as the events are also multiplied as many times as the number of these threads and add extra pressure. Eventually more space is requested from the output network buffers of the source.

5.2 Maritime Monitoring

We now present experimental results on another real-world dataset. This dataset contains trajectories of vessels sailing at sea. We have defined a set of patterns that are similar to the ones presented in [24, 26], which have been constructed with the help of domain experts. We demonstrate the effectiveness of our system which is capable of efficiently processing a dataset that contains trajectories from $\approx 5K$ vessels and covers a period of 6 months in less than one hour.

5.2.1 Dataset Description. A public dataset of 18M position signals from 5K vessels sailing in the Atlantic Ocean around the port of Brest, France, between October 1st 2015 and 31st March 2016 has been utilized [27]. A derivative dataset has been released in [25], containing a compressed version of the original dataset (4.5M signals), as decribed in [24]. Each trajectory in this dataset contains only the so-called critical points of the original trajectory, i.e., points that indicate a significant change in a vessel's behavior (e.g., a change in speed or heading) and from which the original trajectory can be faithfully reconstructed. We processed these compressed trajectories in order to determine the proximity of vessels to various areas and locations of interest, such as ports, fishing areas, protected NATURA areas, the coastline, etc.

5.2.2 Pattern Definitions. We now present a detailed description of the maritime patterns that we implemented, assuming that the input events contain the information described above.

Definition 5.10. High Speed Near Coast: Vessel is within 300 meters from the coast and is sailing with speed greater than 5 knots for at least one message.

$$HSNC := (IsNear(Coast) = TRUE \wedge Speed > 5)^+$$

Definition 5.11. Anchored: Vessel is inside an anchorage area or near a port and is sailing with speed less than 0.5 knots for at least three messages.

$$Anchored := ((IsNear(Port) = TRUE \vee$$
$$WithinArea(Anchorage) = TRUE) \wedge$$
$$speed < 0.5)^{3+}$$

Definition 5.12. Drifting: There is a difference between heading and actual course over ground greater than 30 degrees while the vessel is sailing with at least 0.5 knots for at least three messages.

$$Drifting := (|Heading - Cog| > 30 \wedge Speed > 0.5)^{3+}$$

Definition 5.13. Trawling: A vessel is inside a fishing area sailing with speed between 1 and 9 knots for at least three messages. In addition, it must be a fishing vessel.

$$Trawling := (VesselType = Fishing \wedge$$
$$WithinArea(Fishing) = TRUE \wedge$$
$$speed > 1.0 \wedge speed < 9.0)^{3+}$$

Definition 5.14. Search and Rescue: A SAR Vessel sails with a speed of greater than 2.7 knots and constantly changes its heading for at least three messages.

$$SAR := (ChangeInHeading = TRUE \wedge$$
$$VesselType = SAR \wedge speed > 2.7)^{3+}$$

Definition 5.15. Loitering: Vessel is neither near port nor the coastline while it sails with speed below 0.5 knots.

$$Loitering := (IsNear(Port) = FALSE \wedge$$
$$IsNear(Coast) = FALSE \wedge$$
$$speed < 0.5)^{3+}$$

5.2.3 Recognition Results. We used the 6 patterns defined above as the workload for a number of experiments. Following a similar approach to our fleet management experiments, we avoided replicating the patterns to emulate a real scenario and evaluated them for streams of different event rates.

Figure 5a presents results for the partition-based parallelization scheme. This time the backpressure remains always above 40% for a 6-threaded input source, even with 16 recognition threads. Due to the fact that the CEP operator cannot keep up with the initial rate of the source, the source has to adjust its rate to match the throughput of the CER operator. According to the Flink dashboard, this rate is at most $180K\ e/s$ (with 16 worker threads). This lower throughput of our CEP operator (compared with the fleet management use case) can be attributed to the fact that the patterns are now more complex, as on average they contain more unions, disjunctions and iterations to evaluate for every event. The duration of the dataset is 6 months which translates to roughly 15.5M seconds. Since the total number of input events is about 4.5M, the average event input rate is
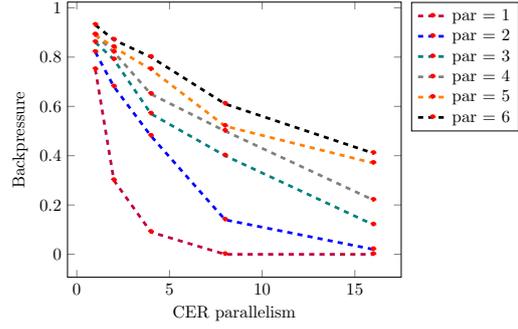
(a) Partition-based backpressure for a workload of 6 patterns. Each dashed line represents a different parallelism of the source stream. Event rate is capped at 180K e/s due to backpressure.
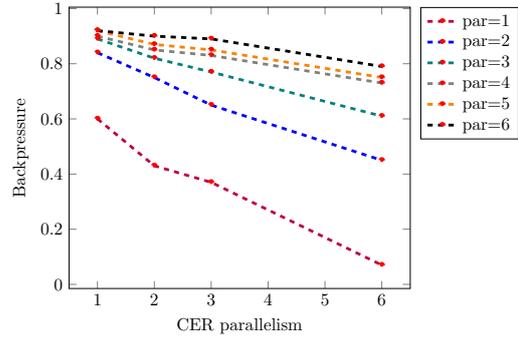
(b) Pattern-based backpressure for a workload of 6 patterns. Each dashed line represents a different parallelism of the source stream. Event rate is capped at 90K e/s due to backpressure.
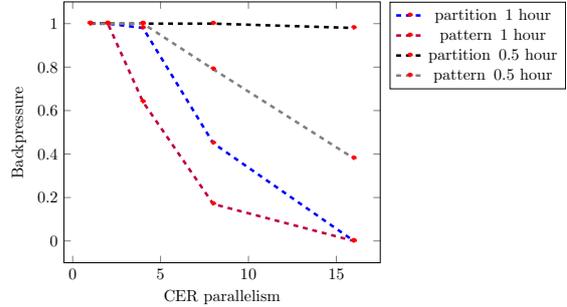
(c) Partition- and pattern-based parallelism for the dataset being replayed in one 1 and 0.5 hours respectively. Workload is 220 patterns of vessels approaching 220 different ports.

Figure 5: Backpressure experiments for the Maritime dataset. The horizontal axis expresses the number of recognition threads.

$15.5M/4.5M \approx 3.5\ e/s$. Comparing the pattern throughput rate with the event input rate showcases again a performance that is 4 orders of magnitude better than the real-time requirements of this use case. The results for pattern-based parallelism are presented in Figure 5b and follow a similar trend. The event rate here is capped at $90K\ e/s$ due to backpressure.

We conducted a second series of experiments with a setting where the number of patterns is naturally high, in order to determine whether pattern-based parallelism offers an advantage in such settings. Consider the following pattern, describing the movement of a vessel as it a approaches a port.

Definition 5.16. *Approaching Port*: Vessel is initially more than 7 km away from the port, then, for at least on message, its distance from the port is between 5 and 7 km and finally it enters the port (i.e., its distance from the port falls below 5 km).

$$Port := (DistanceToPort(Port_X) > 7.0) \land$$
$$DistanceToPort(Port_X) < 10.0) \cdot$$
$$(DistanceToPort(Port_X) > 5.0) \land$$
$$DistanceToPort(Port_X) < 7.0)^+ \cdot$$
$$(DistanceToPort(Port_X) < 5.0)$$

The predicate *DistanceToPort* calculates the distance of a vessel from the port $Port_X$ passed as argument and is evaluated online. If we want to monitor vessel activity around every port in a given area, then we need to replicate this pattern N times, if there are N distinct ports. We would thus naturally have N patterns, which would be almost identical except for the argument passed to *DistanceToPort* ($Port_1$, $Port_2$, up to $Port_N$). For the area of Brest, the total number of ports is 220. We run an experiment with these 220 patterns with partition- and then with pattern-based parallelization. Figure 5c shows the results. Contrary to previous experiments, in this one we used a stream simulator to feed the dataset to our CEP system. This simulator, instead of reading input events from a file and instantly sending them to our engine, has the ability to insert a delay between consecutive events. For example, we can set the delay to be exactly the time difference between two events. This would allow us to re-play the stream as it was actually produced, which would take 6 months for this dataset. We also have the ability to re-play the stream at higher speeds. For these experiments, we re-played the stream at various different speeds in order to determine the "breaking point" of our system. Figure 5c shows the results for two such speeds, where the whole stream was processed in 0.5 and 1 hour, corresponding to a speed-up of $x8640$ and $x4320$ compared to the original dataset. While the CEP operator lags behind the source when it is re-played at half an hour, it is evident that it can process it without any problems when it is replayed at one hour, as both pattern- and partition-based parallelism exhibit 0% backpressure. In fact, pattern-based parallelism performs better in this experiment. This lends credence to our belief that pattern-based parallelism might actually be more suitable than partition-based parallelism when there is a high number of patterns to be processed simultaneously.

6 SUMMARY AND FUTURE WORK

In this paper, we presented Wayeb, a tool for Complex Event Processing, as a means of processing big mobility data streams. We defined a number of detection patterns that are useful in fleet management and maritime monitoring applications. Moreover, we presented implementations of two distributed recognition techniques and compared their efficiency against the single-core version. Our results demonstrate the superiority of partition-based over pattern-based parallelization, when the number of patterns is relatively low. When this number is significantly high, then pattern-based parallelization becomes a viable option. For the future, we intend to combine various distribution techniques and to construct more patterns for the domains presented. Another interesting research avenue would be to compare our automata-based method against other approaches, such as logic-based ones, which have been applied to similar datasets [24, 31].

REFERENCES

[1] [n.d.]. Apache Flink - Stateful Computations over Data Streams. https://flink.apache.org/.
[2] [n.d.]. Apache Kafka. https://kafka.apache.org/.
[3] [n.d.]. Esper. http://www.espertech.com/esper.
[4] [n.d.]. Esperonstorm. https://github.com/tomdz/storm-esper.
[5] [n.d.]. FlinkCEP - Complex event processing for Flink. https://ci.apache.org/projects/flink/flink-docs-stable/dev/libs/cep.html.
[6] [n.d.]. Monitoring Back Pressure. https://ci.apache.org/projects/flink/flink-docs-release-1.9/monitoring/back_pressure.html.
[7] [n.d.]. Siddhi CEP. https://github.com/wso2/siddhi.
[8] [n.d.]. Task chaining and resource groups. https://ci.apache.org/projects/flink/flink-docs-release-1.9/dev/stream/operators/#task-chaining-and-resource-groups.
[9] [n.d.]. WSO2. Creating a Storm Based Distributed Execu-tionPlan. https://docs.wso2.com/display/CEP410/Creating+a+Storm+Based+Distributed+Execution+Plan.
[10] E Alevizos, A Artikis, and G Paliouras. 2017. Event Forecasting with Pattern Markov Chains. In *DEBS*.
[11] E Alevizos, A Artikis, and G Paliouras. 2018. Wayeb: a Tool for Complex Event Forecasting. In *LPAR*.
[12] A Artikis, M Sergot, and G Paliouras. 2015. An Event Calculus for Event Recognition. *IEEE Trans. Knowl. Data Eng.* (2015).
[13] C Balkesen, N Dindar, M Wetter, and N Tatbul. 2013. RIP: run-based intra-query parallelism for scalable complex event processing. In *DEBS*.
[14] P Carbone, A Katsifodimos, S Ewen, V Markl, S Haridi, and K Tzoumas. 2015. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* (2015).
[15] G Cugola and A Margara. 2012. Complex event processing with T-REX. *J. Syst. Softw.* (2012).
[16] G Cugola and A Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* (2012).
[17] L D'Antoni and M Veanes. 2017. The Power of Symbolic Automata and Transducers. In *CAV (1)*.
[18] A Demers, J Gehrke, B Panda, M Riedewald, V Sharma, and W White. 2007. Cayuga: A General Purpose Event Monitoring System. In *CIDR*.
[19] N Giatrakos, E Alevizos, A Artikis, A Deligiannakis, and M Garofalakis. 2020. Complex event recognition in the Big Data era: a survey. *VLDB J.* (2020).
[20] Martin Hirzel. 2012. Partition and compose: parallel complex event processing. In *DEBS*.
[21] N Koutroumanis, G Santipantakis, A Glenis, C Doulkeridis, and G Vouros. 2019. Integration of Mobility Data with Weather Information. In *EDBT/ICDT Workshops*.
[22] M Liu, E Rundensteiner, K Greenfield, C Gupta, S Wang, I Ari, and A Mehta. 2011. E-Cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *SIGMOD*.
[23] Y Mei and S Madden. 2009. ZStream: a cost-based query processor for adaptively detecting composite events. In *SIGMOD*.
[24] K Patroumpas, E Alevizos, A Artikis, M Vodas, N Pelekis, and Y Theodoridis. 2017. Online event recognition from moving vessel trajectories. *GeoInformatica* (2017).
[25] K Patroumpas, D Spirelis, E Chondrodima, H Georgiou, Petrou P, Tampakis P, Sideridis S, Pelekis N, and Theodoridis Y. 2018. Final dataset of Trajectory Synopses over AIS kinematic messages in Brest area (ver. 0.8) [Data set], 10.5281/zenodo.2563256. https://doi.org/10.5281/zenodo.2563256
[26] M Pitsikalis, A Artikis, R Dreo, C Ray, E Camossi, and A-L Jousselme. 2019. Composite Event Recognition for Maritime Monitoring. In *DEBS*.
[27] C Ray, R Dreo, E Camossi, and AL Jousselme. 2018. Heterogeneous Integrated Dataset for Maritime Intelligence, Surveillance, and Reconnaissance, 10.5281/zenodo.1167595. https://doi.org/10.5281/zenodo.1167595
[28] N Schultz-Møller, M Migliavacca, and P Pietzuch. 2009. Distributed complex event processing with query rewriting. In *DEBS*.
[29] L Snidaro, I Visentini, and K Bryan. 2015. Fusing uncertain knowledge and evidence for maritime situational awareness via Markov Logic Networks. *Inf. Fusion* (2015).
[30] F Terroso-Saenz, M Valdés-Vela, and A Skarmeta-Gómez. 2016. A complex event processing approach to detect abnormal behaviours in the marine environment. *Inf. Syst. Frontiers* (2016).
[31] E Tsilionis, N Koutroumanis, P Nikitopoulos, C Doulkeridis, and A Artikis. 2019. Online Event Recognition from Moving Vehicles: Application Paper. *TPLP* (2019).
[32] H Zhang, Y Diao, and N Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *SIGMOD*.