Insights on a Scalable and Dynamic Traffic Management System

Nikolas Zygouras
Department of Informatics and Telecommunications
University of Athens
Greece
nzygouras@di.uoa.gr

Nikos Zacheilas
Department Informatics
Athens University of Economics and Business
Greece
zacheilas@aueb.gr

Vana Kalogeraki
Department Informatics
Athens University of Economics and Business
Greece
vana@aueb.gr

Dermot Kinane
Dublin City Council
Ireland
dermot.kinane@dublincity.ie

Dimitrios Gunopulos
Department of Informatics and Telecommunications
University of Athens
Greece
dg@di.uoa.gr

ABSTRACT

Complex Event Processing (CEP) systems process large streams of data trying to detect events of interest. Traditional CEP systems, such as Esper, lack the required scalability and processing capability to cope with the constantly increasing amount of data that needs to be processed. Furthermore, user defined rules are static so changes in the monitored environment cannot be easily detected. In this paper we investigate the development of a scalable and dynamic traffic management system. Our work makes several contributions: We propose a novel system that combines Esper with a stream processing framework, Storm, in order to parallelize the processing of larger amounts of data. We propose a novel rules' assignment algorithm for distributing Esper rules to the available CEP engines, in a way that maximizes the overall system's throughput. Finally, our system adapts to changes of the environment by processing historical data via Hadoop and dynamically updating the Esper rules based on the generated results. Our work has been evaluated using real data, in several traffic monitoring scenarios for the city of Dublin. Our detailed experimental results indicate the benefits in the working of our approach and the significant increase in the system's throughput when a large number of Esper rules were examined concurrently.

Categories and Subject Descriptors

H.2.4 [**Information Systems**]: Systems—*complex event processing, stream processing*

1. INTRODUCTION

Today there is a large increase in the amount of data that needs to be analysed and processed in real-time in a wide variety of domains, ranging from financial processing [13] to traffic monitoring [8] to healthcare infrastructures [23]. Complex Event Processing systems (CEP) have emerged as a valid solution for analyzing this huge stream of information and detecting events of interest. In a CEP system, user-defined rules process *primitive* events received from a monitored environment in order to detect *composite* phenomena by composing primitive or other composite events using a set of event composition operators. Complex event processing for such Big Data applications is challenging as they need to be able to process high volumes of stream data at low processing latencies.

Traditional CEP systems such as Esper [14] are unable to cope with the current data deluge, mainly because they are based on centralized architectures where the CEP engine receives and processes all incoming events in a single host. Due to this reason, there is a shift in performing the CEP processing in Distributed Stream Processing Systems (DSPS) such as Storm [27], Streams [9] or Spark [3] which are now considered major platforms for data analysis. While such systems provide scalability and fast processing, they lack expressiveness, as the user must provide the actual implementation of the rules that the system has to execute, often needing to express complex tasks and requiring a large amount of code. In contrast, systems like Esper, provide an SQL-like language, EPL (Event Processing Language) for expressing the rules, making it much easier to use and learn. Given the popularity of frameworks such as Storm, Streams and Spark, optimizing the performance of CEP processing in DSPS systems is important, as they do not only provide robust, scalable and reliable solutions to processing fast larger amounts of data, but can also be cost-effective solutions as they can reduce the money paid by users to hosting environments such as clouds. Our architecture uses these frameworks in order to meliorate the value offered from the Big Data systems, scaling the system's *velocity* and *volume*. The system's *value* is optimized as our system is able to support and execute more rules and process larger volume of tuples.

One challenge with rules running in current CEP systems is that they tend to be rather static; this means that their behaviour does not change radically over time. For example, in a traffic management system we may want to be able to detect when a bus is delayed. In most cases this is accomplished with a rule that compares the computed delay for a newly arrived bus trace with a static threshold;

10.5441/002/edbt.2015.65

when this threshold is exceeded an event is triggered [5], [6]. However, using a pre-defined threshold at all times is not beneficial, as the behaviour of the traffic conditions typically change during the course of the day. So, it is of great interest to automatically decide these rules' thresholds. Towards this goal was the work of [25], however it requires a time consuming training phase before it generates the rules that will be used by their CEP system. In contrast, our aim is to dynamically compute the rules' thresholds and change the rules accordingly, in real-time. In our previous work [6] we have illustrated that the Lambda architecture, operating in both batch and stream processing modes, is a promising approach for processing heterogeneous data streams for intelligent urban traffic management. In this work we focus on the **scalability** aspect of our system and illustrate that our system can effectively support multiple concurrently running Esper engines and can **dynamically** adapt to rules' thresholds changes in real-time.

In this work we investigate the development of a CEP system for scalable and dynamic traffic management, that is both powerful and easy-to-use. We provide a system that offers: (i) scalability, (ii) low-latency processing, (iii) ease of use, and (iv) dynamic rule updating to changing system conditions. We focus on the ease of usage because we want the rules running in our system to be easily understood even by non-expert users. Our proposal combines the two approaches (CEP systems and DSPS), exploiting the expressiveness and ease of use of a traditional centralized CEP system such as Esper, and the scalability and fast processing offered by Storm. Supporting dynamic rules is important because it offloads effort from the user as she no longer needs to manually tune the rules' thresholds. Our approach makes the following contributions:

1. We present a novel system architecture that combines Storm, Esper and Hadoop [17], offering a truly scalable and easy-to-use framework for efficient complex event processing. Storm enables us to use a number of Esper engines in parallel, increasing the overall system throughput. Furthermore, by using more engines we are able to concurrently execute multiple Esper rules, further improving the system's performance.
2. We provide algorithms for distributing the Esper rules to the available engines, examining the impact of the assignment on the system's throughput. Our proposed solution aims to balance the load across the engines, so that rules are allocated in a way that all engines process rules with approximately the same amount of input data.
3. We use the Hadoop MapReduce system that is ideally suited for processing historical data. This allows us to compute new thresholds for the running rules and adapt the Esper engines' in real-time for accurate detection of complex events. By using *dynamic* rules we are able to capture the changing conditions of the environment and detect only events of interest.
4. We have implemented our approach and evaluated it using a real traffic monitoring application in the city of Dublin[1]. Our experimental results indicate that our approach is truly scalable, achieves a significant increase in the system's throughput, and can support several concurrently running Esper rules.

2. BASIC COMPONENTS - BACKGROUND

In this section we give an overview of the basic components of our complex event processing framework and discuss some background information and related work.

[1]http://dublinked.com/datastore/datasets/dataset-304.php

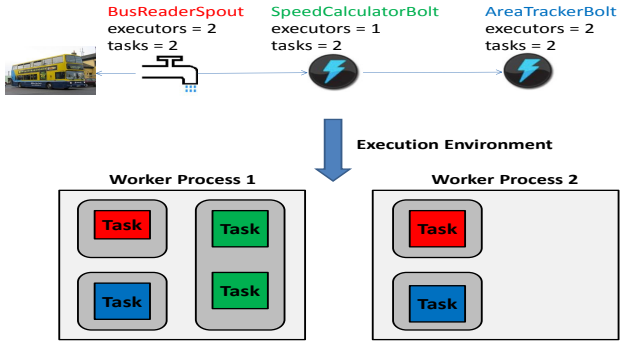


Figure 1: Storm Topology Example

2.1 Basic Framework Components

2.1.1 Storm

Storm [27] has emerged as one of the most commonly used Distributed Stream Processing Engines utilized by major companies such as Twitter [32] and Groupon [16]. It has been successfully employed for processing high volume and intensive workloads in various application domains, where high levels of data throughput and low response latencies are a necessity [26]. Processing massive amounts of data in real-time is achieved by distributing the workload across multiple computers. Applications in Storm are implemented as user-defined topologies. Topologies can be viewed as processing graphs, consisting of nodes that represent user-defined processing operations or primitive event nodes, and edges that represent the streaming of the data. The nodes of a topology graph in Storm can be either *spouts* or *bolts*. Spouts represent the input sources which feed the topology with data, while bolts encapsulate the processing logic. For example, in a CEP application, spouts can be seen as the input sources of primitive or complex events, while bolts are the components that process these events and detect more complex ones.

Users implement the code that will be executed by the Storm components and decide the communication patterns. This essentially represents the subscription of bolts to their input sources. Storm gives the users the capability of deciding how many instances of the implemented bolts' and spouts' code to run in the framework by setting two basic parameters: the number of tasks and executors to utilize. By choosing these parameters, we can increase the parallelism of the topology, making it more scalable. The *executors* parameter (as can be seen in Figure 1) can be used to adjust the number of threads that will execute the processing implemented on the spouts and bolts. The actual processing is performed by the components' *tasks*. Tasks are Java objects containing the user-defined code for the components. Ideally there should be one executor for each task. If the number of tasks is greater than the number of executors, tasks assigned to the same executor are executed in a pseudo-parallel way. In Figure 1, we give an example of a possible assignment of tasks to the corresponding executors in a snapshot of the Storm topology we use for monitoring the traffic conditions in the city of Dublin. Because the SpeedCalculatorBolt has two tasks but only one executor, its tasks are assigned to this single executor. All other components exploit fully the parallelism they can use, thus their tasks are assigned to separate executors.

From an architectural perspective, a Storm cluster consists of a set of physical machines, called *Worker* nodes that are responsible for executing the user topologies, and a Master node called *Nimbus*

that coordinates the execution of the topologies. Once a topology has been allocated to the Storm cluster, its executors are assigned to a set of Java processes (*worker* processes) running on the available nodes. Each node is configured with a fixed number of *slots* that will represent the maximum number of *worker* processes that can execute in it. The assignment of the *executors* to the available *worker* processes follows a simple round-robin approach.

2.1.2 Esper

Esper [14] is a Complex Event Processing (CEP) system, applied to streaming data, that triggers actions when the incoming data satisfy some predefined rules. Esper libraries are available for the Java language such as Esper, and for .NET such as NEsper. Esper keeps all the required data structured in memory making the processing fast. The core of the Esper system is the Esper engine which consists of a set of standing queries (or rules). The plethora of technologies that have been developed in Big Data community require the data first to be saved and then to be processed. Esper, on the other hand, provides real time Big Data analytics as it is a 'NoDatabase' technology meaning that no data has to be saved. Esper stores rules in the Esper engine and when new data arrive checks whether or not these rules are fired. This procedure is continuous, as new arriving data are processed serially and the Esper engine responds in real time if any of the stored events meets the constraints. The triggered events can be pushed further into the Esper engine feeding other rules or sent to their listeners. Listeners are associated with rules and define the actions to be taken when the rule is activated. The user can create queries and add them into the Esper engine. These queries are written in Event Processing Language (EPL) and their syntax is similar to *SQL* queries, with *SELECT, FROM, WHERE, GROUP BY, HAVING* and *ORDER BY* clauses. EPL was designed aiming to be similar to the SQL query language. The main difference between EPL and SQL is that EPL uses views instead of tables. Views are the different operations applied to the incoming data to structure data in an event stream. An example of such operation is the expiry policy for events that specify for how long an event will remain in the event stream. Finally, each EPL query defines a sliding or batch window of the incoming stream that it monitors.

2.1.3 Hadoop

The MapReduce programming and execution model [12], along with its open-source implementation Hadoop [17], has emerged as one of the most widely adopted programming models for processing massive-scale datasets. Hadoop has been utilized in a wide variety of application domains including traffic monitoring, stock-market data analysis and financial trading applications. For traffic monitoring applications, such as the one we study, Hadoop can be used to process and analyze historical data in order to compute and store statistics on the stream data, such as to identify normal traffic conditions in different city areas during the course of the day. In the MapReduce model, each computation, or *job*, is modelled as a sequence of two basic operators: *map* and *reduce*. Jobs are automatically parallelized and executed on the available cluster nodes, these are executed as multiple *map* and *reduce* tasks. The map and reduce tasks have the following specifications:

$map(k_1, v_1) \Rightarrow [k_2, v_2]$

$reduce(k_2, [v_2]) \Rightarrow [k_3, v_3]$

In Hadoop, each *map* task is responsible for processing a distinct chunk of the data stored in its distributed filesystem (HDFS [18]). The output of the *map* phase is partitioned using a hash function into a user-defined number of *reduce* tasks. *Reduce* tasks receive their corresponding input data and invoke the *reduce* method on them. All intermediate data generated by the *map* tasks as well as the final results are stored in HDFS for fault-tolerance, but at the cost of extra processing [29], [33].

2.2 Related Work

Stream processing frameworks such as IBM's Infosphere streams [8], Storm [27] and TUD-Streams [9] have been successfully applied for complex event processing. However they lack an expressive language such the one offered by Esper. So the user is responsible to implement all the components required for detecting complex events. Another recently proposed stream processing engine is Spark [36]. Spark aims to unite the worlds of batch and stream processing offering a common framework for both types of computation. Despite its rise in popularity, it is still limited with respect to the expressiveness of the computations, requiring the user to manually implement multiple processing components.

Authors in [2] have focused on the placement of *worker* processors and *executors* of a Storm topology on the available cluster nodes, with the main goal being the minimization of the inter-node communication. Similar scheduling techniques have been proposed in [21], and [34]. These works are orthogonal to ours and can further enhance the working of Storm, increasing the overall system's performance. In [35], the impact of intra-node communication was examined, and it was illustrated that in order to minimize the intra-process communication overhead the number of *worker* processors should be equal to the number of cluster nodes. We adopted this scheduling policy in our framework to minimize the impact of the intra-node communication in the system's performance.

Traffic monitoring has been a field of great interest in the complex event and stream processing community [8], [28]. However, these works detect events based on statically defined *rules* so any updates to the traffic conditions overtime are not taken into account. In contrast, our proposal computes new thresholds for the *rules* and dynamically updates them. Linear Road benchmark [4] is one of the most commonly used platforms for the evaluation of complex event processing frameworks. Many works such as [10] use this framework for their evaluation. However a real dataset such the one we use, can offer a wider variety of events to detect. A recent city transportation application was proposed in [24]. They implemented an application that enables the sharing of taxi rides in a large city, in a way that is beneficial for both citizens and taxi drivers.

With respect to parallelizing the execution of complex event processing systems, work was mainly done by [7]. They increase the parallelism of a complex event procedure by executing multiple instances of the corresponding Finite State Machine, but each with different proportion of the input data. Their approach differs from ours in the fact that their proposal is limited to sequential *rules* while ours can be applied to all types of Esper *rules*. Our system, from an architectural perspective, is similar to the work proposed in [15]. In their proposal they combine Streams [9] with Esper trying to detect events in a football match. However they use only one Esper engine so they do not exploit the parallelism of the DSPS to improve their system's performance.

Authors in [20] propose a system that supports scalable CEP by using more engines and a rules allocation schema that tries to assign rules to engines based on the similarity of their attributes. Our

approach differs in two aspects: (1) our framework can scale-up automatically via the features provided by Storm, while (2) our proposed rules allocation algorithm takes into account both the attributes' similarity as well as the expected input rate. In [1], they tackle the problem of distributing the processing of primitive events on the event sources by generating multi-step event acquisition and processing plans with the goal to minimize the event transmissions cost. Also in [31], they propose Next CEP, a distributed complex event processing system that optimizes the usage of the available system resources. This work was evaluated on a fraud detection application, applying their rules in streams with credit card transactions. Finally, in [30], a scalable CEP system was proposed that targets industrial infrastructures, specifically, e-energy applications in the cloud. However, they do not provide any rules allocation algorithm for the distribution of the rules to the nodes.

3. TRAFFIC MONITORING

3.1 Setting Up the Problem

This work is focused on developing a novel architecture and techniques for a scalable and dynamic traffic management system. Our objective is to recognize in real time abnormal traffic events, like accidents and traffic congestions in the Dublin City[2]. The Dublin City traffic control receives data from various voluminous sources, including cameras CCTV, static sensors that measure the traffic flow on several junctions and buses that move in Dublin city. It is not possible for humans to monitor this large amount of data, so the development of a system that receives raw data, processes them and alerts automatically in case of an emergency, is required. An example of an emergency situation in Dublin is presented in Figure 2.

Dublin City Council (DCC) Intelligent Transportation Systems department provided us with the main requirements of a traffic monitoring system applied in Dublin city. These requirements are summarized bellow:

- Their main aim is to be able to identify the spatial locations where the traffic behavior from the buses, obtained through streaming, exceeds the expected normal behaviour for that particular location. An example of this is a rule that checks if in three consecutive bus stops, buses traversing them, reported simultaneously delays greater than the expected.
- Another requirement of DCC is to determine the normal behaviour for different spatial locations. In traffic monitoring systems the traffic behaviour varies for different areas of the city. This behaviour also varies for different hours of day and between weekdays and weekends. It is usual to have greater delays and lower speed in the city centre than the suburbs and greater delays in working hours than the weekends. So it is essential to set up different thresholds which will enable us to model normal traffic behaviour for different spatial locations, hours and days.
- Also it is possible these thresholds to change over time; for example if a new road is constructed the thresholds may be relaxed and the system should adapt to these changes.
- Finally the traffic monitoring system should work in real time, be able to process large amounts of streaming data and respond quickly in unusual conditions.

[2]This work was done in the context of an EU-funded project Insight http://www.insight-ict.eu/

Attribute	Description
Timestamp	the time of the measurement
LineId	the line of the bus
Direction	true or false
GPS position	Longitude and Latitude of the bus
Delay	the seconds that the bus is ahead of schedule
Congestion	true or false
Bus Stop	the id of the closest bus stop
Vehicle Id	The value that is used to distinguish different buses

Table 1: Description of the Dataset

Property	Value
Number of buses	911
Size of data	160 MB per day
Number of lines	67
Data frequency	3 tuples/min per bus
Time interval	6am till 3am

Table 2: Dataset Properties

Figure 2: Traffic accident in Dublin city

The traffic monitoring system that we built has been tested on bus data across Dublin city, provided from DCC. Each bus transmits every 20 seconds information about its position and congestions. The description of the data provided by the buses is given in Table 1. The dataset's properties are described in Table 2. In order to get more meaningful information about the traffic conditions we decided to process further the raw data, enhancing them with new features. For each tuple that the buses transmit, we compute the speed of the bus movement and the change in the delay value from its previously received measurement, labelled as actual delay.

3.2 Our System Architecture

The main goals of our work is to provide a scalable and easy-to-use traffic monitoring system. We propose a system architecture (shown in Figure 3) that consists of: (i) a Distributed Stream Processing System (Storm), (ii) a batch processing framework (Hadoop), (iii) a distributed filesystem (HDFS), (iv) a storage medium (MySQL Server) and (v) multiple Complex Event Processing (Esper) engines used for detecting events of interest. Our architecture bares a lot of similarities with the Lambda-Architecture [22], however we differ in that we exploit the expressiveness of CEP engines to support complex rules. In our framework the queries we execute are user-defined Esper rules, and the merging of the real-time and batch views is done by exploiting the capabilities of Esper (we will discuss this in more detail in Section 4.2).

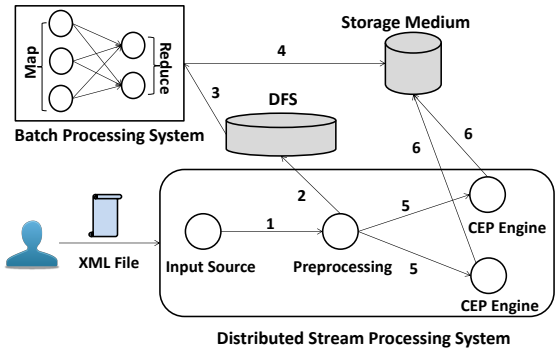


Figure 3: System Architecture

Users in our framework complete an XML file that includes the description of the submitted topology (e.g., spouts, bolts) along with the Esper rules they want to apply to the incoming raw data. We enhanced Storm's library by supporting the creation of topologies via XML. The advantage of this, is that we avoid Storm's standard topology creation procedure, where the user must manually (with Java code) specify how the components should interact with each other. In the simplest case, the user must submit only a spout for specifying the input source along with the rules she wishes to execute. However, for more complex scenarios she can also define extra bolts for pre-processing the raw data before the actual submission to the Esper engines. The pre-processed data before being forwarded to the Esper engines, are stored to a distributed filesystem, HDFS in our case.

Data stored in HDFS are used for computing the threshold values that will be utilized by the Esper rules for detecting complex events. This computation is done by the batch processing layer, in our case Hadoop. We chose Hadoop, since the stored data may increase significantly in size, as new data constantly arrive in our framework. We apply simple MapReduce computations in these historical data and the calculated threshold values are stored to the storage medium in order to be accessible from the Esper engines. In our current implementation the storage medium is a MySQL server but it can easily be substituted by a distributed solution, such as Cassandra [11].

Storm via its scalability features (specifically the usage of more tasks per bolt) enables us to increase the number of Esper engines used for the event detection. This is achieved by increasing the parallelism of the bolt that will be responsible for running the Esper engine. So, if we increase the number of tasks for a bolt, we end up having multiple concurrently running engines. To make full usage of the parallelism, when we increase the number of tasks of the corresponding bolt, we also increase the number of executors in order to run each engine in a separate thread. Furthermore, we allocate the executors into different *worker* processors to make sure that each cluster node will be assigned with the same number of Esper engines.

Using more engines increases the system's throughput and the number of rules that can execute concurrently. Special care must be given on how to allocate the user-defined rules to the available engines (Section 4.2), as we want to fairly distribute the rules, avoiding overloaded situations. Furthermore, rules in our framework should dynamically adapt to new thresholds as threshold values change overtime by the computations performed from the batch processing layer. We examined several techniques (Section 4.3.1) for collecting this information from the storage medium and join them with the streaming data.

3.3 Rules Description

We define rules that allow us to detect events of interest in the traffic data. Finding out where there may be traffic incidents or identifying anomalies in traffic patterns, are examples of events of interest for the Dublin City traffic management system. All events signify certain activities like low speed or increased delays at a particular area. In general the complexity of the rules can vary significantly, as each rule has different characteristics. For example, identifying if a bus is delayed might be simple to detect, while detecting anomalies in traffic patterns might involve computations over multiple simpler events.

We created a generic rule template that checks if the reported attributes from the buses aggregated in different locations, exceed the thresholds. This generic template was selected after discussion with the traffic experts in the Dublin City Council. The rule template has the following parameters: *bus data attribute*, *spatial location* and *window length*. The *bus data attribute* corresponds to the bus data field or fields that the rule checks if they exceed the predefined threshold. Example of these fields are the buses' delay or speed. We monitor different attributes in order to improve our knowledge of the traffic conditions. The *spatial location* is the spatial area that the rule checks for abnormalities. The reason why we selected to create rules for different areas is that traffic jams have spatial extent and in order to identify them we have to search for abnormalities in these areas. The *window length* is the window size of the stream that the rule keeps in memory for processing. In our scenario we compute the average value of all the values in the streaming window in order to compare it with the corresponding thresholds. We used window-based streams because traffic data are very noisy and taking the average value makes them smoother. The generic rule described above has the following format and is fired when the incoming data satisfy the following condition:

$$\wedge f(attribute_i, l, s) > threshold(attribute_i, s)$$

$$threshold(attribute_i, s) = mean(attribute_i, s) + stdv(attribute_i, s)$$

where $attribute_i$ is the *data attribute* that we check, f is the operator that is applied in the stream of data, l is *window length* of the stream that we monitor and s is the *spatial location* in the city map that we monitor. This generic rule's complexity changes overtime as the different *spatial locations* do not have fixed $mean$ and $stdv$.

The EPL code that implements the generic rule template, described above, is presented in Listing 1. The EPL rule contains three streams as it is defined in the $FROM$ clause. The first stream consists of the last event that arrived in the system. The second stream contains the last l values that arrived in the system and have the same $location$ as the last event. The last stream named *thresholdLocation* contains all the thresholds for all the possible locations for different hours of day and for weekdays and weekends. In addition, the streaming data join with this stream in order to retrieve their thresholds. The rule is fired when the average value of a specific $attribute$ is greater than the corresponding threshold, for a specific $location$, $hour$ and day.

Listing 1: Esper EPL Rule template

```
SELECT *
```

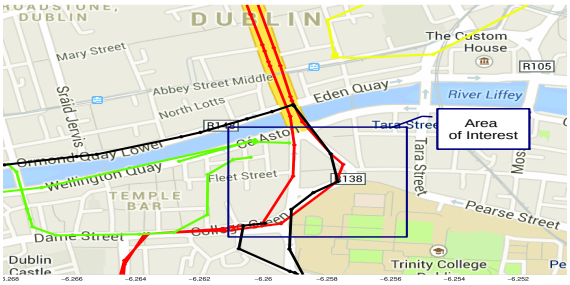


Figure 4: Bus trajectories in a specific area of interest

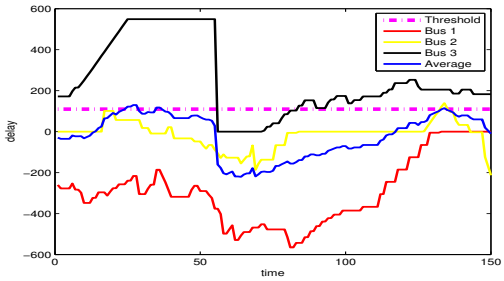


Figure 5: Evolution of delay value for 3 different buses

```
FROM
  bus.std:lastevent() as bd,
  bus.std:groupwin(location).win:length(l) as bd2,
  thresholdLocation.win:keepall() as thresholds
WHERE
  bd.hour=thresholds.hour and
  bd.day=thresholds.day and
  bd.location=thresholds.location and
  bd.location=bd2.location
GROUP BY
  bd2.location
HAVING
  avg(bd2.attribute) > avg(thresholds.attribute)
```

An example of a rule is presented in Figures 4 and 5. Figure 4 shows the trajectories of 3 buses moving in the Dublin city and the corresponding area of interest. The rule checks for abnormalities in this area. The rule is fired when the average delay's value from all the buses that move in the area of interest is greater than the threshold for the particular area. Figure 5 shows the evolution of the 3 buses that are in the bounding box, the average value of delay for the 3 buses and the threshold for this area. The rule is fired when the delay's moving average exceeds the threshold value.

4. DESCRIBING THE COMPUTATION

In order to tackle the traffic monitoring problem we decompose it into three main components:

1. **Off-line Computation.** The computations performed by this component enable us to support *dynamic* rules.
2. **Start-Up Optimization.** This component optimizes the system's parameters according to the user's requirements.
3. **On-line Processing.** The component that is responsible for the actual processing of newly arriving data as well as to detect events of interest.

Our objective in this work is to improve throughput and be able to process as big datasets as possible. We provide algorithms for all three components but we focus on the **Start-Up Optimization** component because this defines the instance of the architecture we run. The first two components are used in order to optimize the performance of the third component.

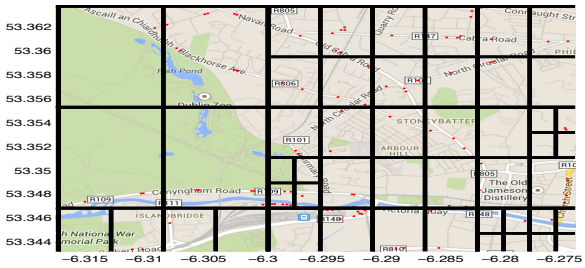


Figure 6: Region Quadtree with bus stops

4.1 Off-line Computation

This component is responsible to prepare the system in order to be initialized and support it in possible future changes. Also it makes our system dynamic as it could change the system's parameters over time and make it to adjust to new, dynamically posed, requirements.

4.1.1 Spatial Indexing

The Dublin City Council requested to be able to monitor city's traffic conditions at different spatial extent, e.g. from small city blocks to whole road lines. For this reason the rules were defined with a hierarchical decomposition regarding the spatial locations. This hierarchical decomposition was not given to us, so we partitioned the map in sub-areas. In order to partition a map there are different approaches that we could apply, including the Region quadtree data-structure, Grids, Voronoi diagrams or even arbitrary shapes that include areas of the city. In our application we utilized the Region quadtree.

The quadtree represents a partition of the space in two dimensions by decomposing each region into four equal sub-regions, and so forth. The region quadtree is created by adding some initial data points to it and then splitting until each region keeps a maximum number of data points. So the resulting tree is not always balanced. In our case the quadtree was created by adding important coordinates of the Dublin city, e.g. main road segments. Because these points are not equally distributed in the city, as can be seen in Figure 6, the regions created by the quadtree are unbalanced. The user decides the spatial extent of the monitoring by specifying in Esper rules, either the layer of the Quadtree she wants to examine or some explicit area of interest.

4.1.2 Bus Stops

Furthermore, we decided to monitor the traffic condition at areas near bus stops in Dublin city. Bus data are noisy, especially when buses report their stops. More particularly, we observed that a specific bus stop is reported at different locations. Also buses reported that they were stopped while they were actually moving. Also nearby bus stops seem to have different ids. We decided to calculate new bus stops and create a tool, that for each line, direction and GPS position, will identify the closest bus stop.

In order to deal with the problem of identifying the bus stops we applied the DENCLUE [19] clustering algorithm in the GPS loca-

Parameters	Values
Output	Rule's Latency
Input 1	length window of a rule, l
Input 2	number of thresholds in the Engine that rule joins with, t

Table 3: Parameters of Single Rule Latency Function

tions, where the buses reported that they just reached the bus stop. Initially DENCLUE added a 2-dimensional Gaussian distribution with $\sigma = 20m$ at each data point and then added all the Gaussians in order to calculate the global density. Then for each data point we identified its local maxima named as density attractor and we kept together all the points that their density attractors were close. These clusters do not consider the bus direction (i.e. a cluster may have bus stops, where buses move in one and the opposite direction). In order to keep the different directions we decided to split the clusters further. Our approach works as follows: We found the average angle that the bus had when it entered in the cluster per line and direction, and then we placed in the same subcluster the bus lines and directions that had similar average angle. Then for each new set of GPS position, line and direction it is possible to find the closest subcluster. For the rest of this paper we will call these subclusters as bus stops.

4.1.3 Supporting Dynamic Rules

Given that we look for abnormalities during the course of the day for different spatial locations, we compute statistics continuously and update the rules accordingly. These statistics are calculated using Hadoop jobs. The jobs are invoked periodically, e.g., every one hour, to compute statistics for the different spatial locations in the upcoming time window, e.g. for the next hour. Specifically, the job calculates the mean and standard deviation of the parameters defined in Table 6 (see Section 5) for the different locations (e.g. quadtree areas or bus stops). In the map phase we retrieve the historical data from HDFS and then emit them to the reduce tasks. The reducers aggregate the parameters' values for the different spatial locations and then compute the mean and the standard deviation. The results are stored in our MySQL server and are retrieved during the online processing to be used as thresholds for the running rules.

4.1.4 Estimate Engine's Latency

A key factor of our analysis is the estimation of each Esper engine's latency. We build a model that takes as parameters the set of rules to run, their characteristics, the number of available cluster nodes and Esper engines, and estimates the latency of each engine. The model's architecture is presented in Figure 7. In order to do this estimation we created three regression functions that are presented bellow:

- **Single Rule Latency Function (Function 1)** This function estimates the latency of a rule that has window l and t thresholds. As we observed that these are the two main components that affect the latency of a rule. The input and the output of this function are presented in Table 3.
- **Multiple Rules Latency Function (Function 2)** The second function estimates the total latency of an Esper engine to process a tuple when we place multiple rules in the same engine. This function takes as input the latency calculated from the first function. If the user inserts rules with different format as ours, Single Rules Latency Function is not reliable, thus we calculate the latency of the rule running in a single engine and then insert in the second function this information. If we place more than 2 rules we will call this function sequentially, e.g. the output of this function will be fed again as its input. The input and the output of this function are presented in Table 4.
- **Engine's Latency Function (Function 3)** Finally the last function is used for estimating the latency of an engine's rules if it is placed in the same cluster node with other engines. The latency of processing incoming events increases if a cluster node is overloaded with many engines. The input and the output of this function are presented in Table 5, where three Esper engines run in the same cluster node.

Parameters	Values
Output	Engine's Latency
Input 1	Latency of rule 1
Input 2	Latency of rule 2

Table 4: Parameters of Multiple Rules Latency Function

Parameters	Values
Output	Latency Engine$_i$
Input 1	Latency Engine$_i$
Input 2	Latency Engine$_j$
Input 3	Latency Engine$_k$

Table 5: Engine's Latency Function

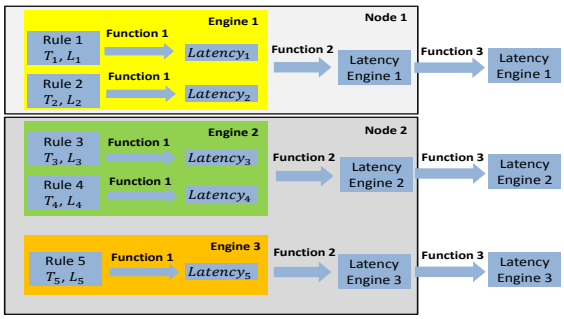


Figure 7: Estimation Model

4.2 Start-Up Optimization

This component is responsible for setting up the system, thus, is executed before the actual Storm topology starts processing incoming data. Initially it collects all the rules that the user chose to run and analyses their requirements. In the next step, the component decides how to allocate these rules to different Esper engines. The allocation is done based on the regression model explained in the Section 4.1.4. Furthermore, the component's optimizations can be invoked periodically (or when new rules are submitted to the framework) to adjust the rules allocation to the current system's conditions.

4.2.1 Rules Partitioning

Balancing the data processed by the Esper engines is one of the key components to improving the overall system throughput. The rules

Rule's Partitioning Component

Input: $Engines$ the set of Esper engines to use, $rule$ the rule that needs to be partitioned
$Region\_Rates \leftarrow$ Retrieve regions' input rates for layer $rule.quadtree\_layer$
Sort $Region\_Rates$ in descending order
for all $engine_i$ in $Engines$ **do**
  $rate[engine_i] = 0$
end for
for all $region$ in $Region\_Rates$ **do**
  $less\_loaded = engine_1$
  $min\_rate = rate[engine_1]$
  for all $engine_i$ in $Engines$ **do**
    if $min\_rate > rate[engine_i]$ **then**
      $min\_rate = rate[engine_i]$
      $less\_loaded = engine_i$
    end if
  end for
  Assign $region$ to $less\_loaded$ engine
  $rate[less\_loaded] = rate[less\_loaded] + region.rate$
end for

Algorithm 1: Rule's Partitioning Algorithm

Rules Allocation Component

Input: N number of Esper engines, $Groupings\_Set$ set of the groupings with their corresponding rules
for all $grouping_i$ in $Groupings\_Set$ **do**
  $scores[grouping_i] \leftarrow$ Estimate $grouping_i$'s rules score with 1 engine
  $engines[grouping_i] = 1$
end for
$N = N - |Groupings\_Set|$
for $j = 1$ to N **do**
  $max\_score = 0$
  $chosen\_group = grouping_1$
  for all $grouping_i$ in $Groupings\_Set$ **do**
    $estimated\_score \leftarrow$ Estimate $grouping_i$'s rules score for $engines[grouping_i] + 1$
    if $max\_score < estimated\_score$ **then**
      $max\_score = estimated\_score$
      $chosen\_group = grouping_i$
    end if
  end for
  $scores[chosen\_group] = max\_score$
  $engines[chosen\_group] = engines[chosen\_group] + 1$
end for

Algorithm 2: Rules Allocation Algorithm

that our system examines detect abnormalities in buses' reported values at different spatial locations. Thus, we partition the rules by sending the data that correspond to different spatial locations into different Esper engines. We follow a partitioning schema that partitions a rule's spatial locations to different engines based on their input rates. The tuples are sent to the appropriate engine according to this schema. We consider as the input rate of a spatial location, the amount of bus traces expected to be processed by the engine in that location. We have some initial knowledge about these rates (e.g. from historical data) and incrementally update them while the application runs. As you can see in Algorithm 1, the rule's examining locations are partitioned in a way that all engines will receive approximately the same aggregated input rate.

4.2.2 Rules Allocation

One of the key components for achieving fast processing of the constantly arriving new primitive events is the allocation of the rules to the available Esper engines. We exploit the hierarchical structure of our rules and provide an efficient assignment of the rules to the engines. Recall that the running rules examine spatial locations. These locations may overlap as one rule may monitor the hole city and other rules some specific roads or neighbourhoods.

Because we have multiple rules per spatial layer and a limited number of engines, we might need to group together rules that belong to different layers. This approach can be beneficial because we avoid data re-transmissions. If we assign each layer in a different engine, newly arriving primitive events must be transmitted to all the engines, limiting the benefits of the Storm's parallelism and adding extra traffic between the cluster nodes. Conversely, if we put all rules examining the second and third quadtree layers in the same grouping, we would not have to sent new events twice as areas of the third layer will be assigned to the same engine with their parents from the second layer. We achieve this by partitioning rules' locations (see 4.2.1) based on the higher possible layer. So in the previous case, we would partition the rules based on the spatial locations belonging to the second layer of our quadtree.

We propose a greedy algorithm (Algorithm 2) that receives as input a possible set of groupings of the different layers examined by the rules, and provides an allocation to the available engines in a way that maximizes a score function. The time required to process the input data for the rules of $grouping_i$ in $Engine_j$ is given by the following formula:

$$time_{i,j} = inputRate_i \times latency_j \tag{1}$$

The score for each grouping will be related to the minimum time required to process its set of tuples in the engines it has been assigned.

$$score_i = \sum_{i=1}^{W_i} w_i \times \min_{j \in Engines}(time_{i,j}) \tag{2}$$

where W_i is the number of rules in $grouping_i$. Also w_i is the weight of a particular rule. The traffic management operator may select that some rules are more important than others. For example, an appropriate policy may be to place higher weights in the rules that take much time to execute.

Our algorithm is not a simple variation of the Bin-Packing problem but the process we follow is a bit more complex. If we place more than one rule in the same engine, the estimation of the engine's latency is not trivial, as it varies for different combinations of rules. For this reason we propose an algorithm that utilizes the regression model for estimating the observed latency when we allocate the rules to multiple engines and then computes the corresponding score via Equation 2. The algorithm first gives in each grouping a separate engine to execute. So if the initial grouping considers the root layer with the second layer together, while the third layer separately, then the algorithm would allocate one engine for the first pair of layers and a second engine for the third layer's rules. Furthermore for each grouping we estimate its achieved score for this initial assignment.

In the next step of the algorithm, we estimate for each grouping its score if we had added an extra engine for that particular grouping. The grouping that leads to the greater score increase is the one that will use the extra engine. This approach enables us to maximize the achievable score without examining all the combinations of layers and engines. In each step we keep the new score estimation for the chosen grouping and increase the number of engines appropriately. We repeat this procedure until all engines have been utilized.

4.3 On-line Processing

This component consists of the Storm topology and the Esper engines. The rules are allocated to these engines via the techniques proposed in the **Start-Up Optimization** component.

4.3.1 Retrieve Batch Generated Data

Esper rules do not have a unique threshold but have different thresholds for different input data (i.e. different bus stop, areas etc). Recall that these values are computed by Hadoop and stored in an SQL server. These thresholds are retrieved using the SQL query that is presented in Listing 2. This query takes as parameter the value s which tunes the value of the threshold as s times the standard deviation away from the mean.

Listing 2: SQL Query that retrieves the thresholds

```
SELECT DISTINCT
  attr_mean+s*attr_stdv as thresholdLocation,
  currentHour,
  dateType,
  areaId1
FROM
  statistics_attribute
```

In order to feed the rules with these thresholds we tested the following three methods:

- **Join with Database** Each new tuple that arrives in an Esper engine will do a join with the database in order to retrieve the corresponding threshold.
- **Create Multiple Rules** Retrieve all the thresholds from the database in advance and for each possible combination create the relevant rule.
- **Add the Thresholds in an Esper stream** Retrieve all the thresholds from the database and add them in a new Esper stream. Each new tuple will join with this thresholds' stream.

4.3.2 Storm Topology

In Figure 8, we saw the Storm topology that we created for the described traffic monitoring application. The application consists of seven components. Newly arrived bus traces are emitted to the topology from the BusReader spout. In our current implementation the traces are stored in csv files so we use this spout for reading the stored data. The PreProcess Bolt is responsible for adding extra information such as the vehicle's speed and direction. These data are forwarded to the Area Tracker bolt which detects the areas in the different quadtree layers where the vehicle currently resides. Each task of this bolt has an instance of the Region Quadtree and queries it to find the areas that the new trace belongs. The BusStops Tracker bolt also adds a new attribute in the examined trace, specifically it adds the bus stop id. Then the data are emmitted to the Splitter Bolt that is responsible to send each tuple to the appropriate Esper Bolt task. As we mentioned above our system consists of many Esper engines located in different tasks of the Bolt. It is crucial to route each bus data tuple to the appropriate Esper engine as each engine examines different spatial locations (Section 4.2.1). Finally the Esper Bolt executes the user-defined *rules* that are used for detecting unusual traffic conditions in the city. We have multiple tasks of this bolt to exploit the inherent parallelism of Storm. Detected events by the EsperBolt are forwarded to the EventsStorer bolt which stores them to a pre-decided storage medium, in our case a MySQL server.

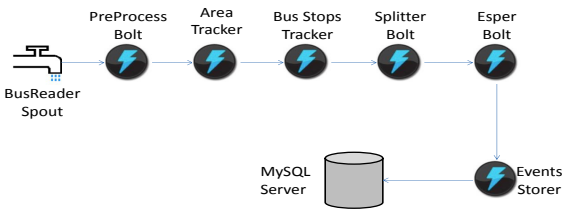


Figure 8: Traffic Monitoring Topology

Parameters	Values
Attribute:	Delay, Actual Delay, Speed, Delay and Congestion, All
Location:	Bus Stops and Quadtree Areas
Window Length:	1, 10, 100, 1000

Table 6: Parameters of the generic rule template and the corresponding values

5. EVALUATION

We have performed an extensive experimental study of our approach on our local cluster consisting of 7 VMs, running on three actual nodes. Each VM had attached one CPU processor and 2 GB RAM. All VMs were connected to the same LAN and their clocks were synchronized with the NTP protocol. We used Storm 0.8.2, Hadoop 1.2.1 and Esper 5.1. We used a separate node in our cluster where the Storm Master process (Nimbus) executed to avoid overloading one of the VMs. In this work, we use the values in Table 6 as parameters of the generic rule template described in Section 3.3. For our evaluation we fed our system with bus traces from the period of 1st to 31st January (4 GB in total) at full speed, so without any delay between the tuples inter-arrivals. We followed this policy to stretch our system's performance, specifically every second our application received $60,000$ bus traces.

We focused on the performance of the bolt that runs the Esper engines, as it is responsible for the more heavy-weight processing and also because it is the part of the topology where our optimizations were applied. Two main **metrics** were considered:

1. The achieved *throughput* in regards to the number of input data that are processed in a fragment of 40 seconds
2. The average *latency* to process a single input tuple. Again we considered a time period of 40 seconds.

To collect these data we enhanced Storm with an extra monitor thread per *worker* processor, that periodically (every 40 seconds in our case) reports these metrics for each bolt's task to the Nimbus node. The Nimbus aggregates these data to compute the final monitor metrics per bolt.

5.1 Regression Evaluation

In order to build the regression model, described in Section 4.1.4, that estimates the latency for an engine if we add in it two sets of rules, we used polynomial regression. Initially we ran several experiments in order to build the appropriate dataset and we splitted it in training and test set. Then we feeded a first and a second order polynomial regression model in this dataset. From the experiments

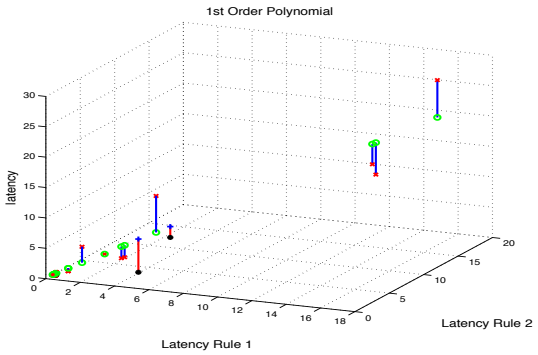
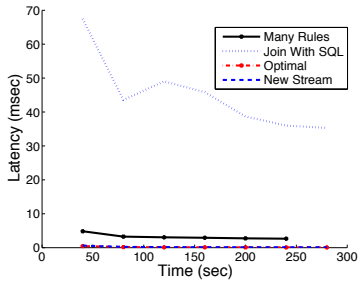
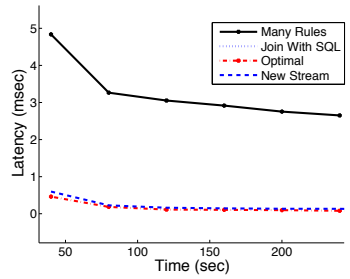


Figure 9: Polynomial for Multiple Rules Latency Function



(a) Observed latency for different retrieval methods



(b) Zoomed observed latency for different retrieval methods

Figure 10: Performance of retrieving location thresholds methods

we identified that the first order polynomial regression has less average absolute error (around 60%) than the second order. This is the reason why we selected to use this model in order to model this function. The identified function is: $0.0077598 \times latency_1 + 2.3016 \times e^{-05} \times latency_2 + 2.4717$. The generated model is presented in Figure 9. We followed a similar approach for creating the other two regression functions.

5.2 Retrieving Results From Storage Medium

We evaluated our three proposed techniques for dynamically retrieving the rules' thresholds from the storage medium, depicting their impact on the observed latency. We also provide results when a static threshold is used. In this case no data needs to be retrieved from the storage medium. This depicts the optimal scenario where we do not have the retrieval overhead. As you can see in Figures 10(a), 10(b), using an inner SQL query for each rule deteriorates the performance of the framework because for each incoming tuple we have to join the tuples' attributes with the ones stored in the storage medium, leading to a significant increase in the latency.

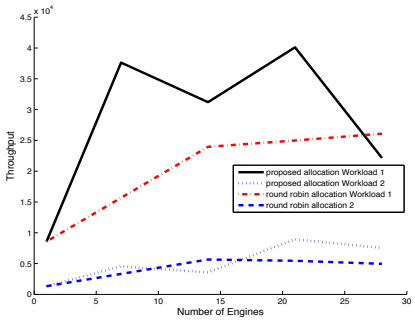


Figure 11: Rules Allocation Throughput Performance

In regards to the two other methods you can see that using a new stream for the thresholds (blue line in the Figures) has latency comparable to the one observed in the no threshold scenario. The multiple rules approach leads to an increased latency because the engines are overloaded with multiple rules, so there is a significant difference compared to the new stream approach as you can see more clearly in Figure 10(b). For the remaining experimental evaluation we used the new stream approach.

5.3 Rules Partitioning

In regards to partitioning the rules to their allocated engines, we compared our proposal with two other approaches:

1. **All Grouping**: Rules' spatial locations are partitioned to the engines similarly to our approach, but newly arrived tuples are emitted to every engine.
2. **All Rules**: All engines have all the rules' locations allocated to them and each incoming tuple is forwarded to the engine that was decided by our partitioning schema.

We compared them with our algorithm when 10 rules are running. Five of the rules examined each of the different attributes (see Table 6) for the bus stops, while the other five monitored the leaves of the quadtree. All rules had 100 tuples in their window length. As you can see in Figures 12, 13, our partitioning proposal achieves a larger increase in the system's throughput, because the system is not overloaded with extra tuples (as the **All Grouping** technique does) and also the engines are not overloaded with rules as in the **All Rules** scenario.

5.4 Rules Allocation

We evaluated the performance of our proposed allocation algorithm, when rules belonging to different quadtree layers must be allocated to the available engines. We used two Workloads based on the attributes and locations described in Table 6. Workload 1 used 1, 10 and 100 window lengths while in Workload 2, rules had 100 and 1000 as window lengths. We compared our algorithm with a simple round-robin approach that considers the rules based on the layer of the quadtree they belong. The algorithm assigns the engines to these layers via a round-robin fashion. As you can see in Figure 11 our algorithm achieves better results because it allocates rules from different layers together avoiding the overhead of retransmissions that occur when the round-robin algorithm is applied. Specifically our algorithm allocated for both workloads all rules together, until fourteen engines were considered. When fourteen engines were used, rules concerning the bus stops were

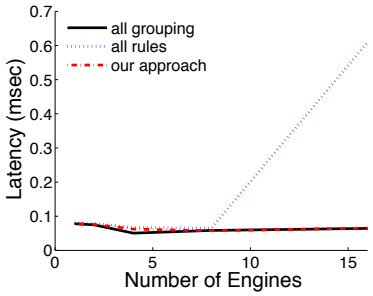


Figure 12: Observed latency for different partitioning approaches

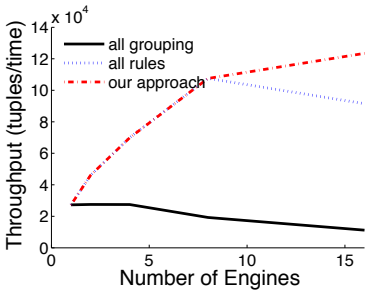


Figure 13: Achieved throughput for different partitioning approaches

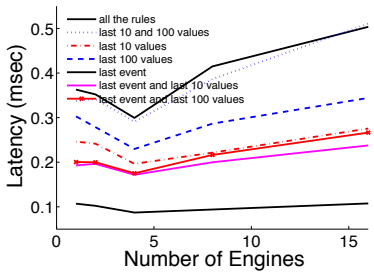


Figure 14: Observed latency for different workloads

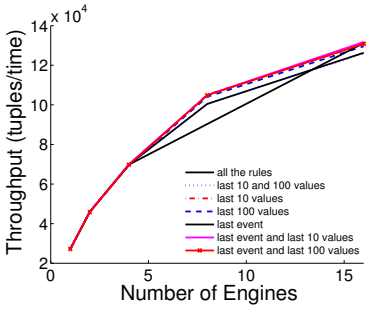


Figure 15: Achieved throughput for different workloads

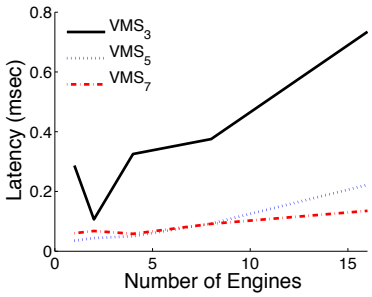


Figure 16: Observed latency for different number of VMs

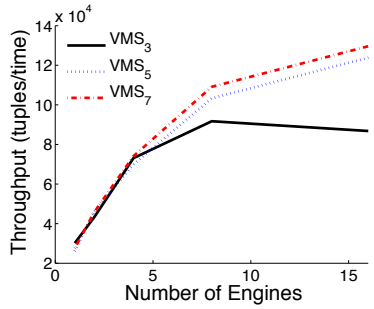


Figure 17: Achieved throughput for different number of VMs

examined in their own dedicated set of engines, and all the other rules were put together to the remaining engines. So our approach maximizes the overlapping between the layers and thus minimizes the required data re-transmissions.

5.5 Different Workloads

We also examined the performance of our system when different workloads are assigned to the available Esper engines. In these experiments we used our proposed allocation algorithm, and examined how our system scales with different workloads. We considered three different workloads based on the rules running in our application. In our evaluation we also examined cases where these workloads were issued concurrently.

- **Last Event**. Consisted of ten rules that kept only one previous tuple in their time window. Five rules examined the attributes values at the bus stops, while the other five monitored the attributes in the leaves layer of the quadtree.
- **Last Ten Values**. Consisted of ten rules that kept ten previous tuples in their time window. Rules had the same attributes and locations as the **Last Event** workload.
- **Last One Hundred Values**. Similarly, this workload consisted of ten rules with the same structure as the other two workloads, only this time we kept one hundred previous tuples in the rules' time windows.

As you can see in Figure 15, our system is able to achieve a steady increase in the overall system's throughput even when we apply all the workloads at the same time.

5.6 Scalability

Finally we evaluated the scalability of our framework when we vary the number of VMs that were used. We examined the framework's performance for 3, 5 and 7 VMs. We used the last workload from the previous section for these experiments. In Figures 16, 17, you can see that when more VMs are available we achieve a steady throughput increase. In regards to latency we can see how overloading the system can lead to its significant increase. For example in the 3 VMs case, using more than four Esper engines leads to a huge increase in the observed latency. Also you can observe that the best results in regards to latency occur when we do not exceed the available processing resources (CPU cores).

6. CONCLUSION

In this paper we presented a novel traffic management system for detecting complex events in the city of Dublin. We proposed a new system architecture that combines Storm, Esper and Hadoop, offering a truly scalable and easy-to-use framework for efficient complex event processing. We provided algorithms for allocating the rules to the available Esper engines and processing historical data in order to be able to support *dynamic* rules. Our experimental results in our local cluster indicate a clear improvement in the system's performance.

7. ACKNOWLEDGMENTS

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thalis-DISFER, Aristeia-MMD, Aristeia-INCEPTION Investing in knowledge society through the European Social Fund,

the FP7 INSIGHT project and the ERC IDEAS NGHCS project.

8. REFERENCES

[1] M. Akdere, U. Çetintemel, and N. Tatbul. Plan-based Complex Event Detection across Distributed Sources. *Proceedings of the VLDB Endowment VLDB Endowment Hompage archive Volume 1 Issue 1, Pages 66-77, August*, 2008.

[2] L. Aniello, R. Baldoni, and L. Querzoni. Adaptive Online Scheduling in Storm. *DEBS*, 2013.

[3] Apache's Spark. `https://spark.apache.org`.

[4] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear Road: A Stream Data Management Benchmark. *VLDB '04 Proceedings of the Thirtieth international conference on Very large data bases*, 2004.

[5] A. Artikis, M. Weidlich, A. Gal, V. Kalogeraki, and D. Gunopulos. Self-Adaptive Event Recognition for Intelligent Transport Management. *IEEE Conference on Big Data, 319-325*, 2013.

[6] A. Artikis, M. Weidlich, F. Schnitzler, I. Boutsis, T. Liebig, N. Piatkowski, C. Bockermann, K. Morik, V. Kalogeraki, J. Marecek, A. Gal, S. Mannor, D. Kinane, and D. Gunopulos. Heterogeneous Stream Processing and Crowdsourcing for Urban Traffic Management. *in Proc. 17th International Conference on Extending Database Technology (EDBT), Athens, Greece, March 24-28, pp. 712-723*, 2014.

[7] C. Balkasen, N. Dindar, M. Wetter, and N. Tatbul. RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing. *DEBS*, 2013.

[8] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. *SIGMOD '10 Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[9] C. Bockermann and H. Blom. The streams framework. *Technical Report 5, TU Dortmund University*, 2012.

[10] I. Botan, P. M. Fischer, D. Kossmann, and N. Tatbul. Transactional Stream Processing. *EDBT '12 Proceedings of the 15th International Conference on Extending Database Technology*, 2012.

[11] Cassandra. `https:/cassandra.apache.org`.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *OSDI*, 2004.

[13] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subscribe Systems. *EDBT'06 Proceedings of the 10th international conference on Advances in Database Technology Pages 627-644*, 2006.

[14] Esper. `esper.codehaus.org`.

[15] A. Gal, S. Keren, M. Sondak, M. Weidlich, H. Blom, and C. Bockermann. Grand Challenge: The TechniBall System. *DEBS '13 Proceedings of the 7th ACM international conference on Distributed event-based systems Pages 319-324*, 2013.

[16] Groupon. `http://www.groupon.com`.

[17] Hadoop. `http://lucene.apache.org/hadoop`.

[18] HDFS. `hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[19] A. Hinneburg and D. A. Keim. An Efficient Approach to Clustering in Large Multimedia Databases with Noise. *KDD*, 1998.

[20] K. Isoyama, Y. Kobayashi, T. Sato, K. Kida, M. Yoshida, and H. Tagato. Short Paper: A Scalable Complex Event Processing. *DEBS '12 Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems Pages 123-126*, 2012.

[21] V. Kravtsov, P. Bar, D. Carmeli, A. Schuster, and M. Swain. A scheduling framework for large-scale, parallel, and topology-aware applications. *Journal of Parallel and Distributed Computing, Volume 70, Issue 9, Pages 983 - 992*, September 2010.

[22] Lambda Architecture. `lambda-architecture.net`.

[23] M. Liu, M. Ray, D. Zhang, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, and I. Ari. Realtime Healthcare Services Via Nested Complex Event Processing Technology. *EDBT '12 Proceedings of the 15th International Conference on Extending Database Technology Pages 622-625*, 2012.

[24] S. Ma, Y. Zheng, and O. Wolfson. T-Share: A Large-Scale Dynamic Taxi Ridesharing Service. *ICDE*, 2013.

[25] A. Margara, G. Cugola, and G. Tamburrelli. Learning From the Past: Automated Rule Generation for Complex Event Processing. *DEBS*, 2011.

[26] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne, and S. Petrovic. Scalable Distributed Event Detection for Twitter. *BigData Conference: 543-549*, 2013.

[27] Nathan Marz's Storm. `https://github.com/nathanmarz/storm`.

[28] K. Patroumpas and T. Sellis. Event Processing and Real-time Monitoring over Streaming Traffic Data. *Web and Wireless Geographical Information Systems Lecture Notes in Computer Science Volume 7236, pp 116-133*, 2012.

[29] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All Roads Lead to Rome: Optimistic Recovery for Distributed Iterative Data Processing. *CIKM*, 2013.

[30] B. Schilling, B. Koldehofe, U. Pletat, and K. Rothermel. Distributed Heterogeneous Event Processing: Enhancing Scalability and Interoperability of CEP in an Industrial Context. *DEBS '10 Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems Pages 150-159*, 2010.

[31] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed Complex Event Processing with Query Rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[32] Twitter. `http://twitter.com`.

[33] J. Urbani, A. Margara, C. Jacobs, S. Voulgaris, and H. Bal. AJIRA: a Lightweight Distributed Middleware for MapReduce and Stream Processing. *ICDCS*, 2014.

[34] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, K.-L. Wu, and L. Fleischer. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. *Middleware Lecture Notes in Computer Science Volume 5346, 2008, pp 306-325*, 2008.

[35] J. Xu, Z. Chen, J. Tang, and S. Su. T-Storm: Traffic-aware Online Scheduling in Storm. *ICDCS*, 2014.

[36] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica. Discretized Streams: Fault Tolerant Streaming Computation at Scale. *SOSP*, 2013.