Dynamic Load Balancing Techniques for Distributed Complex Event Processing Systems

Nikos Zacheilas[1], Nikolas Zygouras[2], Nikolaos Panagiotou[2], Vana Kalogeraki[1], and Dimitrios Gunopulos[2]

[1] {zacheilas,vana}@aueb.gr, Athens University of Economics and Business, Greece
[2] {nzygouras,n.panagiotou,dg}@di.uoa.gr, University of Athens, Greece

Abstract. Applying real-time, cost-effective Complex Event processing (CEP) in the cloud has been an important goal in recent years. Distributed Stream Processing Systems (DSPS) have been widely adopted by major computing companies such as Facebook and Twitter for performing scalable event processing in streaming data. However, dynamically balancing the load of the DSPS' components can be particularly challenging due to the high volume of data, the components' state management needs, and the low latency processing requirements. Systems should be able to cope with these challenges and adapt to dynamic and unpredictable load changes in real-time. Our approach makes the following contributions: (i) we formulate the load balancing problem in distributed CEP systems as an instance of the job-shop scheduling problem, and (ii) we present a novel framework that dynamically balances the load of CEP engines in real-time and adapts to sudden changes in the volume of streaming data by exploiting two balancing policies. Our detailed experimental evaluation using data from the Twitter social network indicates the benefits of our approach in the system's throughput.

1 Introduction

In recent years we observe a significant increase in the need for processing and analyzing voluminous data streams in a variety of application domains, ranging from traffic monitoring [19] to financial processing [4]. In order to analyze this huge amount of data and detect events of interest, Complex Event Processing (CEP) systems have emerged as an appropriate solution. In CEP systems, like Esper[1], users define queries (i.e. rules) that process incoming *primitive* events and detect complex events when some conditions are satisfied. CEP systems are easy to use as most of them offer a query language for expressing the rules.

One significant shortcoming of CEP systems is that they lack in scalability due to their centralized architecture, making them inadequate for applications that require processing large volumes of data streams. On the other hand, Distributed Stream Processing Systems (DSPS) like Infosphere Streams[2], Spark

[1] esper.codehaus.org [2] www.ibm.com/software/products/us/en/infosphere-streams

Streaming[3] and Storm[4] are commonly used for performing scalable and low latency complex event detection. However, current DSPSs lack the expressiveness and ease of use of CEP systems. So combining the two approaches provides a scalable and easy to use framework.

A common approach to increase the system's throughput is to distribute different CEP engines across the cluster nodes, using DSPS [19]. In this work we follow this approach and focus on applications that apply the *key-grouping* partitioning schema for distributing the input data to the CEP engines. *Key-grouping* partitioning assigns the tuples to the appropriate CEP engines based on a specific key attribute of the tuples, which ensures that tuples sharing the same *key* will end up in the same engine. However, when applying a static partitioning schema, which remains the same during the topology's lifetime, it is highly possible to create imbalanced cluster nodes. This usually happens in scenarios where the data load that share the same *key* varies significantly over time. For example, in applications that monitor stock prices (i.e. where the grouping *key* is the stock's name), the number of particular stock transactions may vary significantly during the day. Thus, sudden changes in the system's load and performance are quite common [8]. So it is important to implement techniques that are able to dynamically adapt to such changes in the data load and ensure that the system remains balanced [12][14]. These techniques should consider the size of the data that will be transferred across the engines. Many data re-transmissions result to increased recovery time (i.e. *rebalancing cost*).

Another technique that is widely used in order to cope with the data deluge is elasticity [5]. Elastic systems are able to decide the appropriate number of engines (i.e. scale-out, scale-in) and adapt to changes in the observed load (e.g. unexpected load bursts). In our previous work [18], we have described a novel technique that enables the automatic adjustment of the number of running engines. In order to keep the system's performance stable the load balancing problem and the elasticity problem are *orthogonal*. When the system is unable to process the incoming data, actions that balance the load should be taken firstly and if this fails then scale-out actions should be applied. Furthermore, load balancing techniques can be beneficial when applications run in cloud infrastructures like Amazon's EC2[5]. In such environments users are charged on an hourly-basis, so having overloaded engines can lead to increased monetary cost as they would require more time to process their assigned data.

In this paper we focus on the problem of automatically balancing the load between concurrently running CEP engines. Our work aims at dynamically adapting to unexpected changes in the input load and minimizing the amount of data that needs to be exchanged between the engines so that the state migration overhead is reduced. Our contributions are the following:

- We formulate our problem as an instance of the *job shop scheduling* problem.
- We propose a novel Dynamic Load Balancing (*DLB*) algorithm that balances the engines' load and at the same time minimizes the required state

[3] `https://spark.apache.org`
[4] `http://storm.apache.org/`
[5] `http://aws.amazon.com/ec2/`

migrations. We examine two different policies for deciding which *keys* should be moved.
- We add an extra component in Storm, named *Splitter*, in order to support these load balancing algorithms and handle the necessary state migrations that guarantee the system's consistency.
- Finally, we evaluate our proposals in our local Storm cluster with a Twitter application that performs First Story Detection [11] on incoming tweets. Our experimental results demonstrate that our framework effectively balances the load between the CEP engines and improves the system's throughput compared to other state-of-the-art techniques [12] [14].

2 System Architecture and Model

2.1 System Architecture

We have built our system using Apache's Storm as the DSPS and Esper as the CEP system.

Esper. Esper is one of the most commonly used Complex Event Processing (CEP) systems, applied to streaming data for detecting events of interest. Users can define queries (i.e. Esper rules) via the Event Processing Language (EPL), which bears a lot of similarities with SQL thus it is easy to use and learn. Incoming data are examined and when they satisfy the rules' conditions an event is triggered. The windows of the data streams, based on their expiration policy, could be either *length-based* (for a fixed number of data points) or *time-based* (contain the tuples received during a sliding time window). Esper keeps incoming data in in-memory buffers for the time period needed and then discards them.

Storm. We chose to use Apache's Storm mainly due to its scalability features that allow us to process voluminous data streams [11]. Applications in Storm are expressed as a graph, called topology. The graph consists of nodes that encapsulate the processing logic (implemented by the users) and edges that represent the data flows among components. There can be two types of *components*: *spouts* and *bolts*. Spouts are the input sources in the system which feed the framework with raw data (e.g. simple events). Bolts process these incoming tuples with user-defined Java functions. From an architectural perspective Storm consists of a Master node, called Nimbus, and multiple worker nodes. The user can easily tune the parallelism of the components (spouts/bolts) by adjusting the number of threads (i.e. *executors* in Storm's terminology) that will be used for the components' execution. These threads run on the workers' processors, enabling the topology's distributed execution in the cluster's nodes.

Combining DSPS and CEP. In Figure 1 we present our system architecture. Users decide the components to be used (e.g. spouts and bolts) and the Esper rules to run. We added a special type of bolt called EsperBolt that contains an Esper engine that processes incoming tuples and invokes the user-defined rules. This way we exploit the ease of use of Esper as users need to define only the rules that would execute and the actual evaluation of the rules is applied by the Esper engines. Furthermore, users can set the parallelism (i.e. number of threads) of the

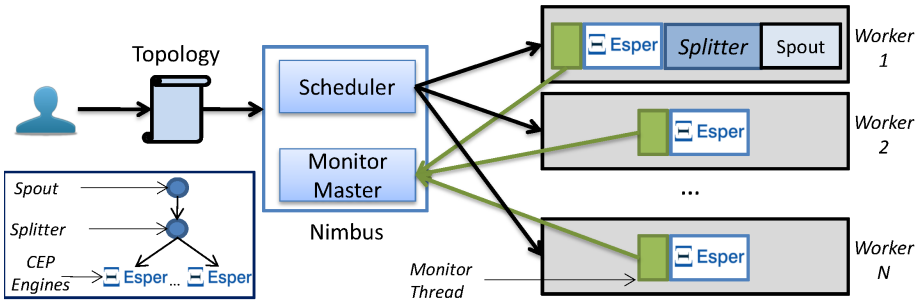


Fig. 1. System Architecture

EsperBolt. In the topology presented in Figure 1, the component that contains the EsperBolt has its parallelism set to N so we have N CEP engines running in our topology, and each one of them executes in a separate processor. Also we added an extra monitor thread per worker's processor to report periodically the performance (e.g. latency, input rate) of its assigned components.

Our system is able to receive and distributively process voluminous sequential data from a wide range of input sources (e.g. Twitter data, bus mobility data and stock prices). In this work we focus on rules that group together tuples based on some common characteristic, named as *key*; tuples that refer to a specific *key* should be transmitted always to the same Esper engine. So we need a partitioning schema that maps the possible *keys* to the available engines. Another challenge arises from the fact that the amount of tuples that correspond to each *key* may vary significantly overtime and this can affect the system's performance.

In many real-world applications, content-aware partitioning is required in order to preserve system's functionality. It is extremely possible to miss events or to identify wrong ones, if tuples with the same *key* are assigned to different CEP engines. For example, in an application that monitors the evolution of different stock prices, the tuples that refer to the same stock should be sent to the same CEP engine. Following this approach, each CEP engine is responsible to process a subset of the input *keys*.

2.2 System Metrics

Each Storm topology is associated with a set of Esper *Engines* that will be responsible for the distributed event processing and a set of $Keys$ that distinctly characterize a group of tuples (i.e. tweet's topics, stock's name). Each Esper rule monitors and identifies events on the set of tuples that share the same *key*.

We define the following set of metrics to measure the system's performance:

- $keys\_map[k]$: a data structure that maps each possible *key* $k \in \{1, ..., |Keys|\}$ to an Esper engine $e \in \{1, ..., |Engines|\}$. As a result, different *keys* will be grouped together in the same Esper engine. Streaming data will be transmitted to the appropriate engine according to this data structure.
- $keys\_load[k]$: represents the amount of incoming tuples that share the same *key* k in a pre-defined time or length based window.

– *eng_load*[*e*]: represents the amount of incoming tuples emitted to the engine *e* in the examined window defined as:

$$eng\_load[e] = \sum_{(k \in Keys | e = keys\_map[k])} keys\_load[k], \forall e \in Engines \quad (1)$$

– *imbScore*(*eng_load*): is a function, defined in Section 3.1, that depicts the system's *imbalance* score given as input the engines' load. The goal of this work is to minimize this metric.

3 Methodology

In this work we focus on the problem of balancing the load among concurrently running CEP engines by determining the appropriate assignment of *keys* to engines that will keep the system's performance steady even in the case of varying *keys* load. Initially, we formulate our problem and describe a metric that measures the system's imbalance. Then, we propose an algorithm, Dynamic Load Balancing (*DLB*), which solves efficiently the problem. Furthermore, we consider two policies, *DLB-L* and *DLB-H*, for determining the *keys* that should be moved in order to balance the load. Finally, we present how we extended our system to support these techniques and appropriately migrate the engines' state to guarantee that our CEP engines report correct events. For the latter, we use a distributed database for storing and retrieving the in-memory tuples of the Esper engines when the algorithm changes the partitioning schema.

3.1 Problem Definition

Our problem can be represented as a variation of a well-known *NP-hard* optimization problem, the *job shop scheduling* problem [3]. More formally our optimization problem can be formulated as follows:

> *Given a set of Keys where each* key $k \in Keys$ *receives* $keys\_load[k]$ *tuples/sec, a set of Engines where each engine* $e \in Engines$ *receives* $eng\_load[e]$ *tuples/sec and a current allocation of* keys *to engines* $keys\_map[\,]$*, our goal is to find a new assignment of* keys *to engines* $keys\_map'[\,]$ *that minimizes imbScore metric and also minimizes the data that will be transferred across the different engines.*

In the *job shop scheduling* problem we have a set of identical machines and we want to schedule a set of jobs in these machines in order to minimize some performance metric (e.g. the total makespan which is the total length of the schedule). In our setting, *Engines* can be seen as the identical machines of the *job shop scheduling* problem, while *Keys* correspond to the jobs that must be assigned into these machines and *imbScore* is the performance metric we want to minimize. Note that in our problem we also consider state migrations when a *key* must move to a different engine, ensuring that the CEP will remain consistent.

Algorithm 1 Dynamic Load Balancing

Input: threshold τ
Output: $keys\_map$
$imbScore \leftarrow measure\_imbalance(eng\_load)$
while $imbScore > \theta$ **do**
  $sort_{DESC}(eng\_load)$
  $e_{min} \leftarrow arg\,min_e(eng\_load)$
  $eng\_load_{temp} \leftarrow eng\_load$
  while $eng\_load.hasNext()$ **do**
    $e_{loaded} \leftarrow eng\_load.getNext()$
    if $e_{loaded} == e_{min}$ **then**
      return $keys\_map$
    $k^* \leftarrow selectKey(policy, keys\_load[\,])$
    $eng\_load_{temp}(e_{loaded}) \leftarrow eng\_load_{temp}(e_{loaded}) - keys\_load(k^*)$
    $eng\_load_{temp}(e_{min}) \leftarrow eng\_load_{temp}(e_{min}) + keys\_load(k^*)$
    $imbScore\_new \leftarrow measure\_imbalance(eng\_load_{temp})$
    if $imbScore\_new < imbScore$ **then**
      $keys\_map[k^*] \leftarrow e_{min}$
      $eng\_load \leftarrow eng\_load_{temp}$
      $imbScore \leftarrow imbScore\_new$
      $break$
return $keys\_map$

As imbalance function we decided to use the relative standard deviation ($RSTD$) among the engines' load which is a commonly used metric for expressing imbalances in a system [6]. We compute the $imbScore$ as follows:

$$imbScore(eng\_load[\,])) = 100 * \frac{std(eng\_load[\,])}{mean(eng\_load[\,])} \tag{2}$$

where $mean()$ and $std()$ are functions that compute the mean and the standard deviation respectively. The higher the relative standard deviation is, the more imbalanced are the CEP engines. When this quantity exceeds a pre-defined threshold we assume that the load is unequally distributed across the engines so a balancing algorithm must be applied in order to rebalance the load across the engines and minimize this metric.

3.2 Dynamic Load Balancing

Our proposed algorithm can be thought as an extension of the LPT algorithm (Longest Processing Time) [3], which is a greedy approximation for the *job shop scheduling* problem. The main difference with the LPT algorithm is that in our case $Keys$ are already assigned into a finite number of $Engines$, but their load changes over time, resulting to significant load imbalances. This will be the result of an increase or decrease of the aggregated volume in some specific engines. When a balancing algorithm applies a new partitioning schema to react to this, then a set of tuples is transferred to the new engine. This ensures the

consistency of the system's state after the rebalancing. A key feature of our approach is its low complexity as it does not need to monitor system resources (*e.g.,* CPU) but focuses on the data instead. We describe below the two policies of our approach.

Dynamic Load Balancing (*DLB*), presented on Algorithm 1, addresses the previously described issues by dynamically changing the partitioning of different *keys* to the available Esper engines. Initially it checks whether the imbalance score (*imbScore*) exceeds a predefined threshold θ. If the set up criterion is not satisfied then iteratively starts the rebalancing procedure. The rebalancing procedure, presented in Algorithm's 1 lines $5-20$, initially sorts in descending order the loads of the different engines, so the engines that receive more data will be examined firstly in order to reduce their incoming load. Also the least loaded engine, e_{min}, is identified in order to transfer load to this engine. Then our algorithm examines the most loaded engine e_{loaded} and checks whether transferring one *key* k^* from e_{loaded} to e_{min} improves the *imbScore*. If the *imbScore* is improved then k^* is transferred from e_{loaded} to e_{min} and this procedure is repeated till the imbalance criterion is satisfied. In case that *imbScore* is not improved e_{loaded} examines the next most loaded engine. Finally if all engines are examined and the balancing criterion is still not met the rebalancing procedure terminates and the best possible found allocation is returned.

Our algorithm supports different policies in order to select the appropriate *key*, k^* that will be transferred from a loaded engine to the least loaded engine, in Algorithm's 1 line 12. Below we describe the two main policies that we considered in this work:

- *Pick heavy loaded (*DLB-H*):* This policy selects the most loaded *key* from the engine e_{loaded} that reduces the imbalancing score when it is transferred to the least loaded engine e_{min}.
- *Pick lightly loaded (*DLB-L*):* This policy selects the least loaded *key* from the engine e_{loaded} and transfers it to e_{min}.

The first policy, *DLB-H*, minimizes the number of *keys* that are transferred along the engines. This is achieved as it tries to balance the system, moving few heavy loaded *keys*. On the other hand *DLB-L* policy may favour the movement of many poorly loaded *keys* to ensure balance. The first policy outperforms the second in the *length-based* streams, where each *key* contains the same amount of data. Thus, the system will recur in a shorter time period, as the number of transferred *keys* is minimized and consequently the size of transferred data to the new engine is minimized. The second policy is able to fine tune the system better as moving small *keys* across the different engines achieves a more balanced system. Since the rebalancing procedure is terminated when the *imbScore* does not exceed a threshold θ, we expect from the two methods to have a similar performance in the *time-based* streams, where the volume of data in the system for each *key* is proportional to the *key*'s load.

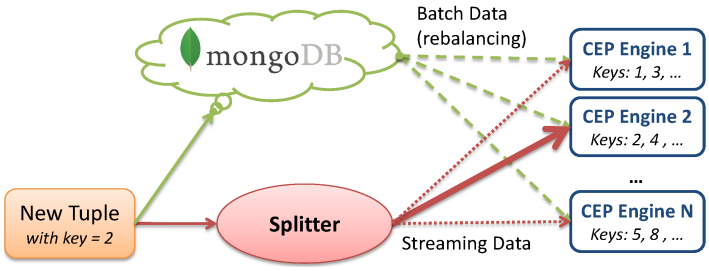


Fig. 2. *Splitter* Component, when a new tuple with *key*= 2 is inserted it routes it to the appropriate CEP engine, also coordinates the rebalancing procedure

3.3 Managing the Engines' States Ensuring CEP's Consistency

Our goal is to be able to support load balance and at the same time guarantee that the detected events will be the same as those detected using the default *key-grouping*(KG) approach. In order to achieve our goals, we added *Splitter*, a new component in our Storm topologies. *Splitter* adjusts the engines' load using the Dynamic Load Balancing algorithm described in the previous and also manages the engines' state in order to ensure that event evidence is preserved by performing content-aware load retransmission. This way we avoid both missing events and false positives that could be caused from the balancing mechanism. *Splitter*, illustrated in Figure 2, is responsible to forward tuples to the appropriate CEP engine based on their *key*. The *Splitter* keeps the *keys_map*[], *keys_load*[] and *eng_load*[] data structures (defined in Section 2.2) and is responsible to update them when new data are inserted into the system. Initially, it determines the engine that will process the incoming tuple using a simple modulo hash function on the tuple's *key*. However, when the imbalance threshold is not satisfied (see Section 3.2) it invokes Algorithm 1 to balance the load between the engines and update the *keys_map*[] data structure.

We store all incoming tuples into a distributed database (e.g. MongoDB[6]) so that the engines can retrieve them when the rebalancing procedure is applied and avoid information loss. Furthermore, *Splitter* is responsible to coordinate the rebalancing procedure, when a *key* k^* is selected to move from an old engine e_{old} to a new one e_{new}. The following steps are followed: (i) e_{old} is informed to remove all the tuples related with k^* that are kept in-memory, (ii) e_{new} is informed that from now on it will be responsible for the processing of k^*'s tuples, (iii) e_{new} retrieves the required data regarding k^* from the distributed database, (iv) e_{new} stores incoming tuples regarding k^* in a local buffer until the necessary data are retrieved and then forwards them to the engine, (v) *Splitter* forwards tuples regarding k^* to e_{new}. In our current implementation, the Splitter component can be the bottleneck as it is possible to be overloaded by the amount of data it receives. As future work, we plan to examine techniques for parallelizing this component and consider the support of multiple input streams.

[6] `http://www.mongodb.org/`

```
SELECT *
FROM Tweet.std:lastevent() LE,
     Tweet.std:groupwin(fingerprint).win:length(H) as TW
WHERE LE.fingerprint = TW.fingerprint
HAVING MAX(TWEET_SIMILARITY(LE.TEXT,TW.TEXT))<=τ
```

Listing 1.1. First story detection rule written as an Esper EPL rule: LE data stream contains the last received tweet while TW contains the last H (set to 500) tweets for each different key. An alarm is fired when the similarity between last received tweet and its closest neighbor sharing the same key is less than a threshold τ (set to 0.2)

4 Evaluation

We have performed an extensive experimental evaluation of our framework in our local cluster consisting of 8 VMs running in two physical nodes. Each VM had attached two CPU processors and 3,072 MB RAM. All VMs were connected to the same LAN and their clocks were synchronized with the NTP protocol. We used Storm 0.8.2, Esper 5.1 and MongoDB 2.6.5. We used a separate physical node where Nimbus runs to avoid overloading the VMs. We evaluated our proposals with a First Story Detection (FSD) [11] application applied on the Twitter data stream. We tested the methods using the default Twitter data order as well as with a modified version that varies the *keys*' load distribution over time using Zipf distribution. In all the experiments described below, 5 Esper engines were used unless stated otherwise. We compared our proposed *DLB* algorithm against two commonly applied techniques, *PKG* [12] and *LPTF* [14]. Furthermore, we also demonstrate how our approach outperforms the *key-grouping* approach (*KG* in Figures 3, 4) which is the default grouping applied in Storm that assigns *keys* to engines using a simple hash function (i.e. $key \ \% \ |Engines|$). The threshold of *imbScore* was set to 15%. MongoDB required 1 ms on average for retrieving a single tweet.

We report results for the following metrics: (1) the system's *throughput* that depicts the amount of tuples per second that have been processed overtime, (2) the *relative standard deviation* of the engines' load which provides insights on how the algorithms balance the engines' load, (3) the *number of complex events* detected by the different techniques, ideally this metric should be equal to the one reported by the *KG* approach as this technique guarantees that tuples with the same *key* will be processed by the same engine and (4) the amount of tuples that are retransmitted by the different techniques in order to balance the system's load. This metric captures the overhead of the rebalancing procedure. It should be *noted* that, in order to test the performance of the proposed methods on extreme conditions, we transmitted the Twitter data to the system with the maximum possible speed, without simulating the original Twitter rate. Also it should be *mentioned* that in order to measure metrics (1), (2) and (4) we run each experiment for 40 minutes, while in order to measure metric (3) the whole dataset was examined.

Table 1. Performance of the Different Techniques with FSD

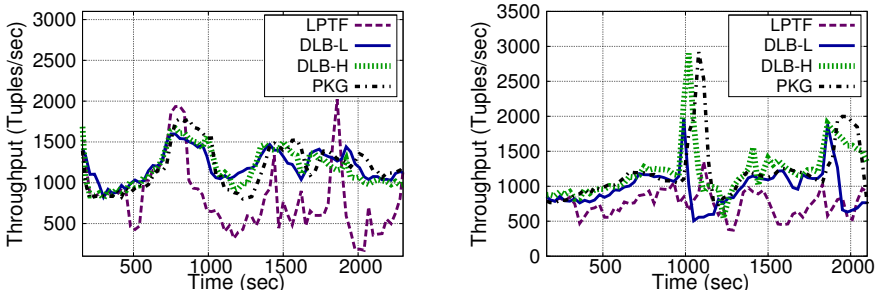
	DLB-L	*DLB-H*	*LPTF*	*PKG*	*KG*
Total Processed Tweets	2,941,246	2,899,076	2,042,201	2,926,853	2,726,628
First Story Events	6,546	6,546	6,546	7,563	6,546
Avg Relative Std (%)	18.34	23.43	14.53	8.6	44.53

Application Description. The FSD algorithm detects the most similar tweet with the current, from a set of the last H received tweets. If the similarity with the most similar tweet is lower than a threshold then this tweet is assumed a novel First Story tweet describing a new event. In order to make the problem tractable and scalable, for each newly received tweet Locality Sensitive Hashing (LSH) is used for identifying the *key* of the tweet [13]. This approach ensures that similar tweets will share the same *key*. The algorithm was translated to an Esper query presented on Listing 1. We fed our system with approximately 2.95 million tweets (8GB) and set the number of *keys* to $4,096$.

As it was mentioned above we selected to transmit the Twitter data to our system with two approaches. The first reads the tweets and transmits them with the default order. The second approach samples the *key* from a Zipf distribution and a tweet with that *key* is transmitted to the system. We selected to vary the Zipf-exponent, ϵ, periodically in order to simulate a use case where the *keys* distribution changes over time. More specifically, initially we started with a low Zipf-exponent, $\epsilon = 0.2$, depicting a rather balanced system, and after five minutes we set ϵ equal to 1.5 creating a highly skewed *key* distribution (approximately 80% $RSTD$ if KG is applied). We kept this highly skewed exponent for 10 minutes and then we repeated the procedure by resetting ϵ to 0.2.

Comparison with other load balancing techniques. Initially we compared our approaches against $LPTF$ and PKG in terms of throughput and number of detected events. Figure 3(a) illustrates the system's throughput when the tuples are read in the default order, while Figure 3(b) depicts the same metric when the periodic Zipf-distribution is applied. It is observed that the two proposed policies (*DLB-L* and *DLB-H*) are able to keep high throughput (approximately 1200 (tuples/sec)) throughout the experiment's execution. On the other hand the $LPTF$ approach suffers from an unstable behavior explained by the large amount of data that are retransmitted along the engines.

More specifically, our policies performed seven retransmissions during the application's execution while $LPTF$ performed nine. However, $LPTF$ in each retransmission moved on average 76% of the *keys* which corresponded approximately to $677,886$ tweets. So a lot of its execution time was spent to move data between the engines. In contrast, *DLB-L* moved at maximum 30% of the *keys* which resulted to $380,000$ tweets. Similar results were exhibited by the *DLB-H* approach that moved at maximum 10% of stored *keys* ($70,400$ tweets). When the *keys* are picked using the periodic Zipf-distribution, Figure 3(b), our approach keeps the throughput steady despite the changes in *keys*' load. When KG was applied with $\epsilon = 0.2$ we observed that $RSTD$ was around 3%, while when



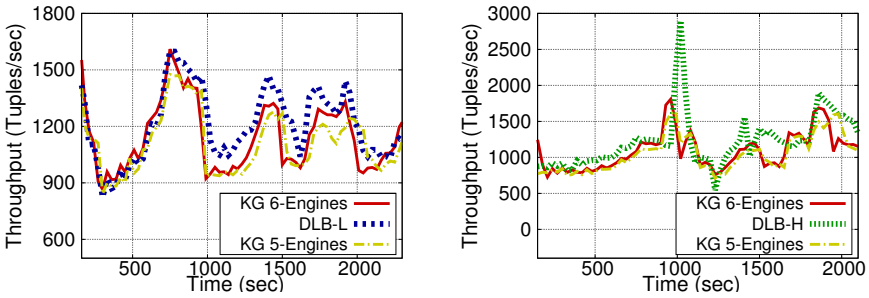
(a) FSD (b) Zipf Distribution

Fig. 3. Throughput comparison with LPTF and PKG

$\epsilon = 1.5$ the $RSTD$ was approximately equal to 77%. $LPTF$ is still penalized by the fact that it moves larger amount of data when a rebalancing occurs and thus has worst performance overtime. More specifically, 12 rebalances occur in the $LPTF$ algorithm moving in each rebalance more than $600,000$ tweets. In this case, *DLB-L* performs ten retransmissions moving 47.5% of the *keys* with $294,881$ moved tweets on average. In contrast, *DLB-L* performs the same number of retransmission as *DLB-H* but moving significantly less data (5% of the *keys* which corresponded to $57,750$ tweets).

In Figures 3(a),(b) we also report the performance of the *PKG* partitioning in terms of throughput. As you can see, its throughput is very stable and similar to the performance of our proposed policies. However, when the system experiences high load (e.g. between $1200 - 1800$ seconds in Figure 3(b)) our proposals are able to maintain higher throughput than *PKG* as they migrate the loaded engines' *keys*. The *PKG* approach processed approximately $2,831,477$ tweets in 40 minutes while *DLB-H* processed $2,795,336$ tweets in the same time period. The main limitation of the *PKG* approach is the fact that it can lead to false positive events. *PKG* may assign tuples that correspond to the same *key* to different engines and thus incorrect events may be detected as the engines' state will not be consistent. More specifically, *PKG* detects in total $7,563$ first story events while our content-aware approach detects $6,546$ events which is the same number of events detected by KG as you can see in Table 1. So *PKG* leads to approximately 13.4% false positive events.

Comparison with scale-out. Finally, in the last set of experiments we examined the performance of our proposals against the *KG* applied by Storm. We report results when *KG* uses 5 and 6 engines as we wanted to point out that scaling-out the system is not always beneficial unless load balancing is also applied. In Figures 4(a),(b) we illustrate the system's throughput overtime. As you can observe, our approach in the unmodified app outperforms *KG* when it uses 5 engines and has comparable throughput with *KG* using 6 engines. Between $800 - 1700$ seconds there is a large increase in the relative standard deviation of the engines' load reaching up to 70% when *KG* is used. *DLB-L* balances the load between the engines keeping it around 20% in this time period and achieves higher throughput. Also as we report in Table 1, in the end of the



(a) FSD (b) Zipf Distribution

Fig. 4. Throughput comparison against KG using more engines

experiment, *DLB-L* has processed 2,941,246 tweets while *KG* with 6 engines processed 2,790,362 tweets and KG with 5 engines 2,726,628.

Similar results were observed in the Zipf-distribution scenario. As you see in Figure 4(b), when the system is not loaded ($\epsilon = 0.2$) using *KG* with six engines outperforms our approach; however, when the exponent is increased (i.e. $300 - 900$, $1200 - 1800$), the relative standard of the engines' load reaches up to 80% and thus the throughput deteriorates. In contrast, *DLB-H* keeps the load small, at around 40%, and thus outperforms the other approaches in these time periods. Finally, when the experiment has finished, *DLB-H* has processed 2,795,336 tweets, *KG* with 6 engines processed 2,653,789 tweets while *KG* with 5 engines processed 2,586,169 tweets.

5 Related Work

Recent works that examine the load balancing problem in stream processing systems have been described in [12, 14]. The dynamic load balancing technique proposed in [12], named Partial Key Grouping (*PKG*), considers the usage of two hash functions for determining two streaming operators and assigns the tuple to the least loaded operator. *PKG* could not be applicable for CEP as it is possible to identify false positive events, as it can send tuples with the same *key* to different engines. More specifically, in CEP systems that use *key-grouping*, each *key* should be emitted to the same engine which contains in memory the previously received tuples with this *key*. In [14] the authors propose the usage of the Longest Processing Time First (*LPTF*) algorithm in offline data and then apply the detected partitioning when the system runs in real-time. *LPTF* is a commonly used greedy approximation of *job shop scheduling* problem that assigns the most loaded *key* to the least loaded engine and repeats this procedure for all the *keys*. The main limitations of this approach are the fact that it requires an offline training phase.

A recent work presented on [10] focused on balancing the load in distributed cache systems. The authors follow an iterative greedy approach which ensures that no node is overloaded by redistributing the node's *keys* to the least loaded

one aiming to balance RAM and CPU usage. Another commonly applied technique for balancing the load in DSPS is load shedding [17] which discards some of the incoming data in order to keep the load steady. However, it can lead to significant information loss. In our previous work [19] we examined the feasibility of developing a large-scale event processing system for a traffic monitoring application. Our approach used historical data and applied a static partitioning schema such that all engines receive approximately the same amount of input. The StreamCloud system described in [7] also supports content-aware load partitioning and load balancing. If the latter fails it moves to a new configuration using its elasticity features. In [16], the authors propose a distributed CEP system that applies query rewriting techniques for optimizing the usage of the system resources. In contrast, our system aims at balancing the load without examining the rules specific characteristics. Furthermore, in [8], authors focus on minimizing the cost of moving the system to a new configuration that utilizes more operators' instances. They focus especially on the latency caused from a system rebalancing suggesting that the shift should be made under latency constraints.

Finally, there has been significant prior work in order to achieve fault-tolerance in distributed stream processing systems [1, 2, 9]. The goal of these works is to minimize the tuples' latency when a crash occurs in the system by exploiting either active replication or upstream backups. The authors in [2] proposed a checkpointing approach to expose the internal operator state. A recent work [9] proposes a hybrid approach by adaptively switching between the two fault-tolerance mechanisms based on the current workload characteristics. Finally, much work has been done in regards to automatically determine the appropriate number of stream processing components like [5, 15] and our previous work [18]. These proposals are *orthogonal* to ours, as our aim is to balance the load among the engines and only if this is not possible, increase the system's resources.

6 Conclusions

In this paper we presented a novel framework that automatically balances the system's load, preserving the system's throughput at high rates. We proposed a balancing algorithm for automatically partitioning incoming tuples to the available CEP engines. Our goal was to keep the CEP engines balanced overtime in regards to the tuples they process and at the same time keep the rebalancing cost, due to data movements, low. Our detailed experimental evaluation in our local cluster indicated a clear improvement in the system's throughput when the proposed techniques were applied. For future work, we plan to extend our framework by enhancing its fault-tolerance and remove possible limitations of the MongoDB in the rebalancing procedure.

7 Acknowledgments

This research has been financed by the European Union through the FP7 ERC IDEAS 308019 NGHCS project and the Horizon2020 688380 VaVeL project.

References

1. Brito, A., Fetzer, C., Felber, P.: Multithreading-enabled active replication for event stream processing operators. SRDS, Niagara Falls, New York, USA (2009)
2. Castro Fernandez, R., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD, New York, NY, USA (2013)
3. Coffman, E.G., Bruno, J.L.: Computer and job-shop scheduling theory. John Wiley & Sons (1976)
4. Demers, A., Gehrke, J., Hong, M., Riedewald, M., White, W.: Towards Expressive Publish/Subscribe Systems. EDBT, Munich, Germany (2006)
5. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. Parallel and Distributed Systems, IEEE Transactions on 25(6), 1447–1463 (2014)
6. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Handling Data Skew In MapReduce. CLOSER, Noordwijkerhout, The Netherlands (2011)
7. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Soriente, C., Valduriez, P.: Streamcloud: An elastic and scalable data streaming system. Parallel and Distributed Systems, IEEE Transactions on 23(12), 2351–2365 (2012)
8. Heinze, T., Jerzak, Z., Hackenbroich, G., Fetzer, C.: Latency-aware elastic scaling for distributed data stream processing systems. DEBS, Mumbai, India (2014)
9. Heinze, T., Zia, M., Krahn, R., Jerzak, Z., Fetzer, C.: An adaptive replication scheme for elastic data stream processing systems. DEBS, Oslo, Norway (2015)
10. Jia, Y., Brondino, I., Peris, R.J., Martínez, M.P., Ma, D.: A multi-resource load balancing algorithm for cloud cache systems. Proc. of the 28th Annual ACM Symposium on Applied Computing (2013)
11. McCreadie, R., Macdonald, C., Ounis, I., Osborne, M., Petrovic, S.: Scalable distributed event detection for twitter. BigData, Santa Clara, CA, USA (2013)
12. Nasir, M.A.U., Morales, G.D.F., García-Soriano, D., Kourtellis, N., Serafini, M.: The power of both choices: Practical load balancing for distributed stream processing engines. ICDE, Seoul, Korea (2015)
13. Petrović, S., Osborne, M., Lavrenko, V.: Streaming first story detection with application to twitter. Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics pp. 181–189 (2010)
14. Rivetti, N., Querzoni, L., Anceaume, E., Busnel, Y., Sericola, B.: Efficient key grouping for near-optimal load balancing in stream processing systems. DEBS, Oslo, Norway (2015)
15. Schneider, S., Hirzel, M., Gedik, B., Wu, K.L.: Auto-Parallelizing Stateful Distributed Streaming Applications. PACT, Minneapolis, MN, USA (2012)
16. Schultz-Møller, N.P., Migliavacca, M., Pietzuch, P.: Distributed Complex Event Processing with Query Rewriting. DEBS, Nashville, Tennessee, USA (2009)
17. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying fit: Efficient load shedding techniques for distributed stream processing. VLDB, Vienna, Austria (2007)
18. Zacheilas, N., Kalogeraki, V., Zygouras, N., Panagiotou, N., Gunopulos, D.: Elastic complex event processing exploiting prediction. BigData, Santa Clara, CA, USA (2015)
19. Zygouras, N., Zacheilas, N., Kalogeraki, V., Kinane, D., Gunopulos, D.: Insights on a Scalable and Dynamic Traffic Management System. EDBT, Brussels, Belgium (2015)