

DIsCO: DynamIc Data COmpression in Distributed Stream Processing Systems

Nikos Zacheilas, Vana Kalogeraki

▶ **To cite this version:**

Nikos Zacheilas, Vana Kalogeraki. DIsCO: DynamIc Data COmpression in Distributed Stream Processing Systems. 17th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2017, Neuchâtel, Switzerland. pp.19-33, 10.1007/978-3-319-59665-5_2 . hal-01800129

HAL Id: hal-01800129
https://inria.hal.science/hal-01800129

Submitted on 25 May 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

DIsCO: DynamIc Data COmpression in Distributed Stream Processing Systems

Nikos Zacheilas and Vana Kalogeraki

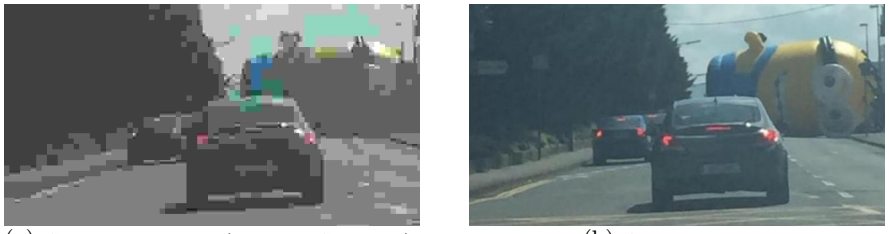
{zacheilas,vana}@aueb.gr
Athens University of Economics and Business, Greece

Abstract. Supporting high throughput in Distributed Stream Processing Systems (DSPSs) has been an important goal in recent years. Current works either focus on automatically increasing the system resources whenever the current setup is inadequate or apply load shedding techniques discarding some of the incoming data. However, both approaches have significant shortcomings as they require on the fly application reconfiguration where the application needs to be stopped and re-uploaded in the cluster with the new configurations, and can lead to significant information loss. One approach that has not yet been considered for improving the throughput of DSPSs is exploiting compression algorithms to minimize the communication overhead between components especially in cases where we have large-sized data like live CCTV camera reports. This work is the first that provides a novel framework, built on top of Apache Storm, which enables dynamic compression of incoming streaming data. Our approach uses a profiling algorithm to automatically determine the compression algorithm that should be applied and supports both lossless and lossy compression techniques. Furthermore, we propose a novel algorithm for determining when profiling should be applied. Finally, our detailed experimental evaluation with commonly used stream processing applications, indicates a clear improvement on the applications' throughput when our proposed techniques are applied.

1 Introduction

In recent years we observe a growing need for supporting complex real-time processing of "big data". Many systems need to process large volumes of live data to detect events of interest in real-time. For example, in a traffic monitoring application it is necessary to inform the city's authorities for events like traffic congestion or accidents [19] as they occur. Similarly, healthcare applications [11] receive input from multiple sensors to detect unusual behavior in the patients' conditions. In order to be able to analyze such a high volume of data, novel distributed systems such as Storm [14], Spark [21] and Flink [4] have been proposed that enable us to perform scalable and low latency complex event detection.

One important challenge in such systems is to support high throughput during the applications' execution despite changes in the data size or the input rate. In the literature, diverse techniques such as elasticity [9] and load shedding [16]



(a) Compressed image (0.01 JPEG quality) (b) Original Image

Fig. 1. Image degradation due to *JPEG* compression.

have been proposed for solving this problem. Elasticity schemes like [9] and [19], automatically increase the amount of system resources (*i.e.,* stream processing operators or components) in order to adapt to sudden load spikes. Such techniques have significant shortcomings: (a) often DSPSs do not support this feature so applications need to be stopped and re-uploaded in the cluster with the new configurations, and (b) approaches such as load shedding [16], [8], automatically drop incoming data when load spikes occur; this penalizes the results' accuracy as many tuples will not be processed and thus important events of interest may be lost.

A technique that has not been fully exploited in these settings is the use of data compression. Data compression can reduce the impact of large-sized data in the components' communication time and thus enables the system to process data faster, increasing the system's throughput. However, applying compression is not always beneficial as it creates additional processing overhead as tuples need to be compressed/decompressed before the operator processes them. In recent years, we observe a plethora of novel compression algorithms [2] or commonly used compression libraries such as LZ4 [13], Zip [22] and Snappy [15] which are applied in the distributed system's domain [5]. However, in current systems the compression algorithm must be manually provided by the users and cannot change during the application's lifetime. Furthermore, it is not trivial for the end-user to determine how useful a compression technique is, when it should be applied, and which compression algorithm is most appropriate to maximize the system throughput. In general, compression techniques can be divided in two major categories, *lossless* and *lossy*. The main difference is that *lossless* techniques enable the perfect reconstruction of the input data and therefore the decompressed data will be the same as the initial data. In contrast, *lossy* techniques include the class of data encoding methods that use inexact approximations and partial data discarding, in order to represent the content. So these techniques lead to data degradation and possibly to inaccurate results if they are applied in event detection applications.

In the following application example we demonstrate that the compression ratio of *lossy* techniques needs to be carefully chosen, taking into account the impact of the compression on the results' accuracy. Our application (described in more details in Section 4) receives as input CCTV camera images streamed in real-time in Dublin city and utilizes a simple image similarity algorithm [12] against historical images which depict normal and abnormal traffic conditions.

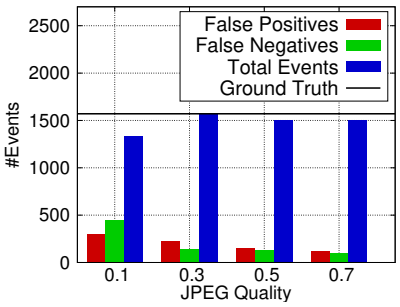


Fig. 2. JPEG's quality metric impact on the results' accuracy.

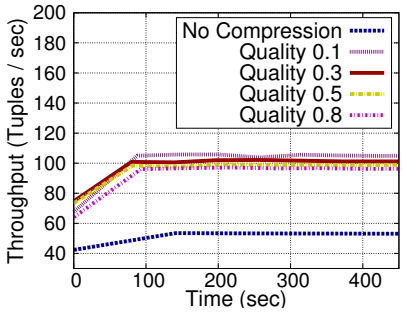


Fig. 3. Throughput using *JPEG* with varying quality.

More specifically, for each incoming image we find the most similar historical image and based on the characterization of the latter (*i.e.,* whether it depicts normal or abnormal traffic conditions) we alert the traffic authorities. In Figures 1(b), 1(a) we illustrate how the *JPEG quality* metric affects the image in terms of visibility. As it can observed when we use very low *JPEG* quality the image is not visible (*i.e.,* in Figure 1(a) the obstacle that fell on the road is blurred) and therefore the application is not able to detect the actual accident that happened (*i.e.,* depicted in Figure 1(b)).

So the *JPEG* quality affects the accuracy of the results and this can be clearly shown in Figure 2 where we illustrate the number of detected events and how many of them are false positives and false negatives. As it is shown in the figure, when we use low compression quality the false positives and negatives events increase due to the degradation of the images. However, when low compression quality is applied it increases the system's throughput due to the smaller image sizes. This is clearly illustrated in Figure 3 where we display the application's throughput using varying *JPEG* quality.

This trade-off between the applications' *accuracy* and the system's *throughput* needs to be taken into account when we determine whether a lossy compression technique like *JPEG* should be applied. Furthermore, it is not trivial for the end-user to understand when compression is beneficial and how different compression algorithms can affect these two metrics. So our aim in this work is to provide a framework that is able to provide answers to the following questions:

1. When is it beneficial to compress the incoming data of a DSPS?
2. Which algorithm should be used for the compression?
3. How the chosen algorithm will be applied efficiently in a distributed streaming environment where the overhead of compression should be minimal?

We propose a novel framework, *DIsCO*, that executes on top of Apache Storm and enables the automatic compression of incoming streaming data for the users' applications. Our approach monitors the application's performance and automatically adjusts the compression algorithm that should be applied taking into account the impact of the compression on the observed throughput and the results' accuracy. The key contributions of this work are as follows:

- We apply a profiling technique that aims to determine when compression is beneficial for Storm applications and which algorithm should be applied. In our framework we consider well-known compression algorithms like *Zip*, *LZ4*, *Snappy* and *JPEG*. Our technique has as goal to find the compression algorithm that balances the trade-off between the application's throughput and the results' accuracy. Our approach works adaptively, where the profiling algorithm is re-invoked when the performance of the application in terms of throughput and results' accuracy has significantly decreased.
- We enhance the Storm system by adding special threads that are part of the Storm's components (*i.e.,* spouts/bolts) and perform the compression/decompression procedure in parallel. This way we minimize the compression/decompression overhead (*i.e.,* due to the time required for compressing and decompressing the tuples) on the application's throughput.
- We conduct an extensive experimental evaluation in our local cluster using different applications that process both text and image data. More specifically, we used a traffic monitoring application that performs image similarity on Dublin's CCTV camera reports, an application that periodically crawls 5 major news sites and searches for traffic incidents and a Twitter First Detection application [20] which processes the Twitter stream and detects first story events. Our experimental results indicate the benefits of our approach and illustrate that we can automatically determine the compression technique that should be applied during the applications' execution.

2 Preliminaries and System Model

In this section we provide a brief description of Apache Storm and provide the key parameters of our approach.

2.1 Preliminaries

Storm is one of the most widely used DSPS mainly because it provides low end-to-end tuples' latency and is widely used in a wide range of application domains including traffic monitoring [19] and Twitter analysis [20]. In Storm, the logic of a stream processing application is packaged into a Storm *topology*; a graph whose nodes are operators that encapsulate the processing logic and edges model data flows among operators. Storm uses a Master-workers architecture where the Master node (*i.e.,* Nimbus) orchestrates the execution of topologies in the available workers. In the Storm terminology nodes are called *components* and the unit of information that is transferred between components is referred to as a *tuple*. Users can define two types of components: (i) *spouts* which are the input sources of the topology, and (ii) *bolts* that encapsulate the processing logic, performing operations such as filtering, correlating and transforming tuples.

In general, there are two main approaches for processing streaming data either on a per tuple basis or in *mini-batches*. The first approach has the lowest

per tuple latency, as tuples are immediately forwarded to the downstream components. However, this approach can increase the communication cost and thus affects the system's throughput [6]. On the other hand, when mini-batches are used, tuples are stalled until the mini-batch is considered ready for further processing [21]. Usually, time (*i.e.,* 1 second has elapsed since the creation of the mini-batch) or size (*e.g.,* emit the mini-batch when it comprises 50 tuples) based criteria are applied for determining when the mini-batch should be forwarded.

Mini-batches are expected to improve the system's throughput but they add overhead on the per tuple's latency as tuples need to wait until the mini-batch is considered ready for processing. We have enhanced the Storm API to support the processing of mini-batches for two main reasons: (1) mini-batches improve the topology's throughput [7], and (2) they enable us to parallelize the compression/decompression procedure and thus minimize its overhead on the application's throughput (*i.e.,* see Section 3.2 for more details). Finally, we did not consider frameworks like Apache Spark [21] and Apache Flink [4] that use only mini-batches for the data processing as we wanted to be able to support the per tuple processing offered by Storm to support applications with very strict time requirements like stock market applications.

2.2 System Model

In this section we define the parameters of our approach. For each Storm topology the user provides the set of *Spouts* that comprise the topology's input sources and the set of *Bolts* which are the topology's processing components. Furthermore, the user provides $CompressAlgos$ which is the set of different compression algorithms that can be applied for compressing the streaming data that are exchanged between the topology's components. In our framework, we support some well-known algorithms (*e.g., Snappy*) but the user can provide also custom implementations. Finally, the user defines the $batchThr$ parameter which controls the frequency with which monitor reports will be sent to Nimbus. For example, if this parameter is set to 10 then every 10 processed mini-batches the tasks will send a monitor report to Nimbus.

When the topology processes $batchThr$ mini-batches, Nimbus receives a monitor report regarding the performance of the compression algorithm c that is currently utilized. More specifically, it receives the $throughput_c$ metric which depicts the number of processed tuples per second when c is applied. Based on the compression algorithm that is utilized we end up with different values for this metric as each algorithm has different effect on the communication cost and also on the time required for the compression/decompression procedure.

Furthermore, lossy techniques penalize the accuracy of the application as they modify the input data. For example, when JPG compression with low quality is applied, the image degrades and some of its characteristics are not visible. In order to take into account this fact, we also measure the number of false positive (*i.e.,* $falsePos_c$) and false negative (*i.e.,* $falseNeg_c$) events when $batchThr$ mini-batches have been processed by a lossy compression algorithm c. Moreover, we compute the number of true positive (*i.e.,* $truePos_c$) and true negative

events (*i.e.,* $trueNeg_c$). In our traffic monitoring application (*i.e.,* described in Section 1), false positive events occur when we mistakenly report that we have traffic congestion. This happens because the image resolution has degraded and the similarity algorithm points out a historical image without traffic congestion as the most similar one. The inverse problem occurs with the false negative events. For lossless techniques, the $falsePos_c$ and $falseNeg_c$ metrics are equal to zero as these techniques guarantee the perfect reconstruction of the input data and thus when they are applied we detect the exact same events as with the case of no compression.

3 Methodology

In this section we describe the basic components of our framework.

3.1 Profiling

The first component of our approach is the use of profiling for determining if compression should be applied and which algorithm to use. Profiling techniques are commonly used for determining the appropriate configuration parameters to be utilized in distributed processing systems [7]. For example profiling has been efficiently applied in the context of Spark applications to determine the appropriate number of nodes that should be allocated to the applications [17]. The benefit is that they can capture the system conditions (*e.g.,* throughput) which can be extremely useful in streaming environments like the one we consider where conditions vary over time.

Our technique receives as input a list of the possible compression algorithms and a threshold on the number of mini-batches that will be processed by each algorithm. In this work we consider three well-known lossless compression techniques (*i.e, Zip*, *Snappy* and *LZ4*) and one lossy compression algorithm (*i.e, JPEG*). The *JPEG* algorithm is examined only when we have image data to be processed while the other techniques are used mainly in case of text data. Moreover, *JPEG* compression depends on the quality metric that determines the compression ratio, so when this compression algorithm is considered the profiling algorithm receives as input the quality metrics that need to be examined. For example, if we want to evaluate three possible values for the quality metric (*i.e.,* 0.1, 0.5, 0.9) then the profiling algorithm will consider three different variations of the *JPEG* algorithm, one for each quality metric.

The basic idea of our algorithm is to execute for a fixed number of mini-batches each possible configuration (*i.e.,* compression algorithm), compute the throughput and the results' accuracy for each compression technique, and then choose the most appropriate technique. Moreover, the profiling technique also examines these two metrics when no compression algorithm is utilized in order to determine whether compression is beneficial. For the results' accuracy we take into account the fact that lossy compression algorithms can lead to false

positive and false negative events (as we indicated in Section 1). More formally, we compute the results' accuracy via the following Formula:

$$acc_c = \frac{truePos_c + trueNeg_c}{truePos_c + trueNeg_c + falsePos_c + falseNeg_c}, \forall c \in CompressAlgos \quad (1)$$

The acc_c metric captures the percentage of false positives and false negatives produced when c is utilized. When lossless techniques are applied those two metrics are equal to zero therefore we have accurate results as acc_c equals 1. However, when lossy compression algorithms are considered we expect a decrease in the accuracy as more false positives and false negatives events will be reported and thus the denominator in Equation 1 will increase. In contrast, we expect better throughput when lossy techniques are applied as smaller-size data are transferred between the components. To balance the throughput with the results' accuracy, we introduce a utility score function as follows:

$$utility_c = \hat{throughput_c} * w + acc_c * (1 - w), \forall c \in CompressAlgos \quad (2)$$

where $w \in [0, 1]$ is a weight given by the user based on where he wants to put more emphasis, *i.e.,* the application's throughput or the accuracy of the results. Furthermore, $throughput_c$ is normalized so that it is in the same range as acc_c.

The goal of our profiling algorithm is to identify the compression algorithm that maximizes Equation 2. The algorithm runs in the Nimbus node as it requires information from all the topology's components. More specifically, our technique consists of the following steps:

1. Initially the profiling algorithm sends to the topology's spouts all the compression algorithms that need to be examined and a threshold on the number of mini-batches to emit for each compression algorithm.
2. The algorithm waits for the monitor reports (*i.e.,* described in Section 2.2) from the bolts that receive and process the data.
3. Upon receiving all reports, the profiling algorithm computes the utility scores of the different compression algorithms using Formula 2. Then it detects the compression algorithm that has the largest utility score (*i.e.,* this indicates that the compression algorithm balances better the trade-off between throughput and accuracy) and informs the spouts that they should use this compression technique from now on.

In order to take into account dynamic changes in the system's condition, we keep monitoring the utility score and if we observe that its value drops significantly we re-apply our profiling algorithm. The technique we used for determining when the profiling algorithm should be re-applied is described in Algorithm 1. Nimbus continues to receive monitor reports and computes their utility scores. When a certain number of reports have been gathered, we compare them with a previous report in terms of utility scores. More specifically, we compute the *tan* of the line that is drawn between the utility scores of the new and the previous report (*i.e.,* Line 5 in Algorithm 1). We use the *tan* metric because it provides an indication of how much the utility score has changed (decreased or increased) from the previous report [3]. If the computed *tan* is positive we have an increase

Algorithm 1 Trigger Algorithm

1: **Input:** *prevTan*: the previous tan that we have computed, *prevReport*: the previous utility score, *monitorReports*: the monitor reports that will be checked for a significant decrease in their utility scores, θ: the threshold used for determining when we have a significant decrease.
2: **Output:** *applyAlgorithm*: a Boolean variable that will determine if the profiling algorithm should be re-applied.
3: $applyAlgorithm \leftarrow true$
4: **for** $(report \in monitorReports)$ **do**
5: $\quad tan \leftarrow \frac{report.getUtility() - prevReport.getUtility()}{report.getTime() - prevReport.getTime()}$
6: $\quad$ **if** $(tan > 0 \;||\; |\frac{tan - prevTan}{prevTan}| < \theta)$ **then**
7: $\quad\quad applyAlgorithm \leftarrow false$
8: $\quad\quad prevReport \leftarrow report$
9: $\quad\quad prevTan \leftarrow tan$
10: $\quad\quad$ **return** *applyAlgorithm*
11: $prevTan \leftarrow tan$
12: $prevReport \leftarrow monitorReports.getLast()$
13: **return** *applyAlgorithm*

in the utility score and thus we stop the search as there is no point to re-apply the profiling algorithm. However, if it is negative (*i.e.,* the utility score has decreased) we must examine whether the difference between the computed *tan* and the previous one (*i.e., prevTan*) exceeds a user-determined threshold θ. More formally, we examine whether the following condition is true:

$$|\frac{tan - prevTan}{prevTan}| > \theta \tag{3}$$

If the condition is true for all the reports that Nimbus has received, then we re-apply the profiling algorithm. Essentially, the *tan* difference helps us identify whether the utility score has decreased from its previous value significantly (*i.e.,* based on the θ parameter).

The performance of the algorithm depends on the θ threshold and the number of monitor reports that we consider. Using a large threshold we expect fewer re-invocations of the profiling algorithm as it will be harder to satisfy the re-invocation condition (*i.e.,* Equation 3). Furthermore, the number of reports that we use in Algorithm 1 also affects the re-invocations as when we use multiple reports it will be harder to satisfy the condition in all of them so we expect less re-invocations of the profiling algorithm. We evaluated how these two parameters affect the re-invocations in our experimental evaluation (*i.e.,* see Section 4). In general, we observed that we need at least three reports to be sure that we have a significant change in the utility score and thus re-applying the profiling technique is beneficial. In contrast, using only one report leads to multiple invocations of the profiling algorithm due to short-term fluctuations of the utility score and this can penalize the application's throughput as when the profiling algorithm is re-applied we have to examine again all the compression algorithms.

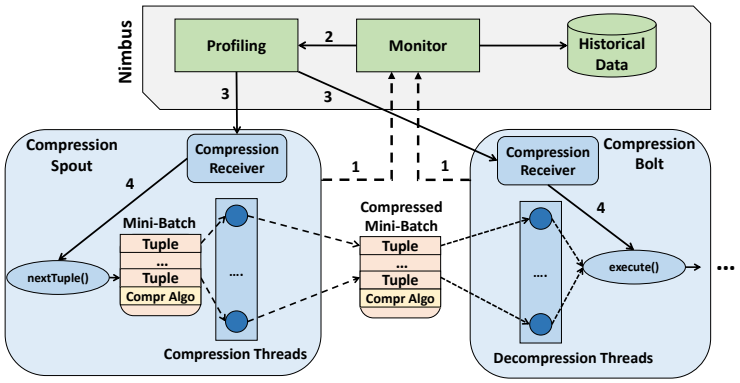


Fig. 4. Implementation Details.

3.2 Parallel Data Compression/Decompression

In Figure 4 we illustrate how the compression algorithms are applied on the mini-batches to be processed. The benefit of utilizing mini-batches is that they enable us to parallelize the compression/decompression procedure. We use multiple threads that run in parallel to perform the decompression whenever the processing component receives a mini-batch. The decompressed data are kept in a mini-batch which will be used for feeding the component with the input tuples. A similar procedure is followed when we compress the data after the output mini-batch is complete (*i.e.,* reached the required batch size) and needs to be emitted to downstream components. In this case the mini-batch is forwarded to multiple concurrently running threads for the compression.

The number of compression/decompression threads can be provided by the user. In most cases the number of threads will be fewer than the number of tuples in the mini-batch so we have to distribute the tuples that comprise it to the available threads. In order to balance the work among the threads we follow a round-robin approach for assigning the tuples to the compression/decompression threads. More formally, assuming that we have N threads, T tuples in the mini-batch and each tuple t in the mini-batch has an id $id_t \in [0, T-1]$ then the thread that will compress tuple t is computed as: $thrId = id_t \% N$, where $thrId$ will be the id of the thread that will be responsible for compressing (or decompressing) tuple t. This simple technique balances the load between the threads and minimizes the compression/decompression overhead.

4 Implementation and Evaluation

Implementation. We have implemented our framework [1] as a module of Apache Storm and in Figure 4 we illustrate the components we have added in order to support dynamic data compression. More specifically, we have added two extra

[1] `http://rtds.aueb.gr/index.php/software/`

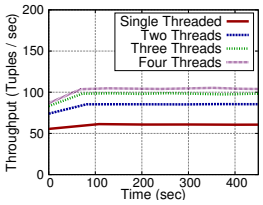
components on Nimbus that are responsible for auto-tuning the compression algorithm that is used by the topology's processing components. The *Profiling* component is responsible for invoking the profiling algorithm described in Section 3.1 and for adjusting the compression algorithm of the Storm components' (spouts or bolts). The second component (*i.e., Monitor* in Figure 4) monitors the performance of all the topology's components. More specifically, *Monitor* receives reports from the bolts' tasks whenever the threshold of processed mini-batches has been reached (*i.e., batchThr* parameter in Section 2.2), computes the utility score (*i.e.,* Equation 2) and informs the *Profiling* component about the new value.

In order to be able to exploit the auto-compression features we offer, users must extend two abstract classes. More specifically, the *CompressionSpout* class must be extended by the users' spouts while the *CompressionBolt* class must be extended by the users' bolts. Each instance of these classes uses a special thread for receiving the compression algorithm that should be applied. Furthermore, these classes create the compression/decompression threads (*i.e.,* see Section 3.2) that are used for minimizing the computation overhead. Users still have to provide the implementation of *nextTuple* (*i.e.,* for spouts) and *execute* (*i.e.,* for bolts) methods in order to be able to utilize their processing components in their topologies.

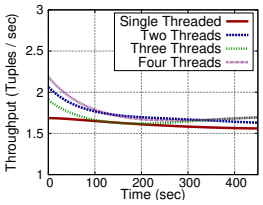
Evaluation Setup. We have evaluated our approach in our local 8 nodes cluster. Each node had attached 8 CPU processors and 16 GB RAM. All nodes were connected to the same LAN and their clocks were synchronized using the NTP protocol. We implemented our proposals on top of Storm 0.10.2 and used a dedicated Nimbus node to avoid overloading one of the nodes. We considered the following applications for examining the performance of our approach:

- **Traffic Monitoring Application**: This application receives as input live CCTV data from Dublin city and detects events by invoking a simple image similarity algorithm (supplied by LIRE framework [12]) against historical images depicting normal and abnormal traffic conditions. More specifically, for each incoming image we find the most similar historical image and based on the characterization of the latter we inform the traffic authorities.
- **URL Crawling**: The application crawls web pages and detects keywords in them. More specifically, we crawl well-known sites (*e.g.,* `https://news.google.ie/`) and try to detect events like accidents in Dublin city.
- **Twitter First Story Detection (Twitter FSD)** [20]: This application retrieves data from the Twitter Streaming API and detects tweets that correspond to new events (*e.g.,* a traffic accident in Dublin city).

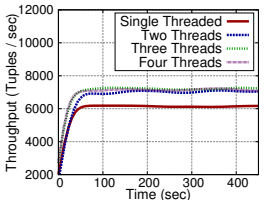
Our goal was to consider applications that process both image and text data. The difference between the two text processing applications is the size of the data. For the URL crawling application we have larger input data (*i.e.,* approximately 600 kB) therefore compression may be beneficial while for the Twitter application the tweets are usually small-sized (*i.e.,* less than 200 bytes) so compression may penalize the application's performance.



(a) Traffic Monitoring



(b) URL Crawling



(c) Twitter FSD

Fig. 5. Parallel Compression Results.

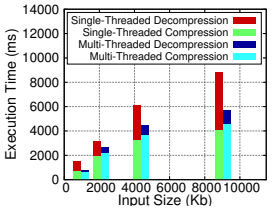


Fig. 6. Impact of data size on compression/decompression time.

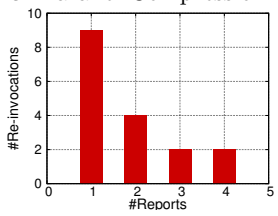


Fig. 7. Impact of reports on the number of profiling algorithm's invocations.

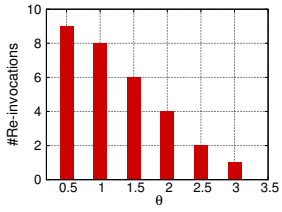
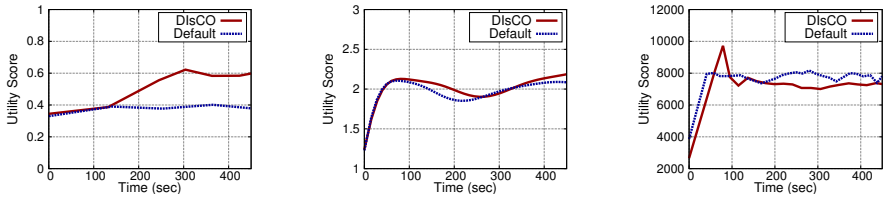


Fig. 8. Impact of θ on the number of profiling algorithm's invocations.

Performance of parallel compression/decompression. In the first set of experiments we illustrate the benefits of performing the parallel compression/decompression procedure. More specifically, we compare our multi-threaded compression technique varying the number of threads. For the traffic monitoring application we used *JPEG* compression with quality 0.1 while for the other two applications we used *Zip* compression (similar results were observed with the other compression algorithms but we do not display them due to lack of space). As can be observed in Figures 5(a), 5(b) and 5(c) using more threads improves throughput as long as we do not exceed the available CPU cores on the nodes. We argue that using at maximum four threads for the compression/decompression procedure is a valid choice when we have 8 core nodes in the cluster. Furthermore, we wanted to evaluate how the compression overhead (*i.e.,* in terms of execution time) is affected by the size of the data that we want to compress/decompress. We used synthetic test data for this experiment and we applied the *Zip* compression algorithm. As we illustrate in Figure 6, our multi-threaded technique (using 4 compression/decompression threads) is able to keep the overhead low (*i.e.,* less than 6 seconds) even when we consider data larger than 8 MB.

Profiling re-invocation. In the second set of experiments (*i.e.,* Figures 7 and 8) we perform a sensitivity analysis on the two parameters that affect the performance of our proposed re-triggering mechanism (*i.e.,* Algorithm 1 in Section 3.1). We considered as input the utility scores of the Twitter FSD application when no compression algorithm is applied (see the *Default* approach in Figure 9(c)) and we examined how the number of monitor reports and the θ parameter affect the number of times that the profiling algorithm will be re-applied.

As we illustrate in Figures 7 and 8 both parameters influence the number of times we re-invoke the profiling algorithm with the one that affects the most being the number of monitor reports to consider for the evaluation. In Figure 7



(a) Traffic Monitoring (b) URL Crawling (c) Twitter FSD

Fig. 9. Utility scores comparison.

we set θ to 0.5 while in Figure 8 we used 2 reports. As it can be observed in Figure 7, if we use more than 4 reports the number of re-invocations is minimized. In contrast, when we use only one monitor report for the comparison, we end up invoking constantly the profiling algorithm. For the rest of the experiments we set θ to 0.5 and use 3 previous reports.

Dynamic compression evaluation. In the last set of experiments we evaluated the applicability of our framework to detect the correct compression algorithm that should be utilized. In Figures 9(a), 9(b) and 9(c) we illustrate the utility score when our framework is applied against the default approach (*i.e.,* *Default* in the Figures) that does not apply any compression algorithm. The weight of the utility score was set to 0.5. As it can be observed, *DIsCO* is able to maximize the utility score in all applications. For example, in case of the traffic monitoring application our framework exploits *JPEG* compression and it is able to guarantee that the impact on the results' accuracy will be minimal. *DIsCO* decides to compress the images using the *JPEG* compression algorithm with quality 0.3. So we have less than 200 false positives and negative events as it can be observed in Figure 2 in Section 1.

For the other two applications the utility score depends solely on the throughput metric as we considered only lossless compression techniques (*i.e.,* for this reason the utility score in Figures 9(b) and 9(c) is not scaled between 0 and 1). The URL crawling application exploits the *Snappy* compression algorithm which is able to achieve better throughput than the *Default* approach. In contrast, in the Twitter FSD application, *DIsCO* detects that it is not beneficial to use a compression algorithm as the improvements in the communication cost are negligible compared to the time required for compressing/decompressing the tweets. However, because *DIsCO* samples all the possible compression algorithms, in this case the *Default* approach has better throughput.

5 Related Work

In recent years we have observed a plethora of stream processing frameworks including Spark [21] and Flink [4]. Despite their popularity, there has been little work in exploiting data compression. One recent proposal comes from the Spark community with the implementation of a novel distributed filesystem called Succint [1] which keeps the data in a compressed form using Arrays of Suffixes (AoS) and enables the processing of the data in this form. However, the wide adoption

of such system requires time as the majority of the data are stored in either HDFS or distributed databases like MongoDB or Cassandra. So an approach like ours that performs on-the-fly compression on the data streams is beneficial to the system's performance.

Previous work exploiting compression for improving the energy efficiency in Hadoop clusters was done in [5]. Our work differs in, that, we examine the problem in a distributed stream processing setting trying to maximize the observed system throughput. Authors in [8] propose the use of an unsupervised learning technique for detecting patterns in the incoming data flow and minimize the amount of emitted tuples by not transmitting tuples that will not contribute on the query's results. In our approach we decided to avoid such load shedding techniques to minimize the information loss. Novel frameworks like [18] and [10] have been proposed for performing analysis on streaming data using a compressed representation of the input dataset. Authors in [18] propose the use of dictionaries for keeping the incoming data and process them in this compressed form while in [10] the authors describe the use of compressed buffer trees (CBTs) for keeping the in-memory data. We could exploit such techniques in *DIsCO* as users can easily plugin their custom compression algorithms.

6 Conclusions

In this paper we present *DIsCO*, a novel auto-tuning framework on top of the Apache Storm whose goal is to balance the trade-off between application's throughput and results' accuracy by dynamically deciding whether data compression would be useful in the streaming applications and which compression algorithm would be the most appropriate. Our approach is able to efficiently adjust the compression algorithm to be utilized during the application's execution, and by exploiting the use of mini-batches it can further maximize the system's throughput and minimize the compression/decompression overhead. In our experimental evaluation on our local cluster using well-known stream processing applications, we demonstrate the benefits of our approach and illustrate a clear improvement on the applications' performance.

Acknowledgment

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. Agarwal, R., Khandelwal, A., Stoica, I.: Succinct: Enabling queries on compressed data. In: NSDI, Oakland, CA. pp. 337–350 (2015)
2. Bicer, T., Yin, J., Chiu, D., Agrawal, G., Schuchardt, K.: Integrating online compression to accelerate large-scale data analytics applications. In: IPDPS, Cambridge, MA, USA. pp. 1205–1216 (2013)

3. Boutsis, I., Kalogeraki, V.: Location privacy for crowdsourcing applications. In: UbiComp, Heidelberg, Germany. pp. 694–705 (2016)
4. Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V., Tzoumas, K.: Apache flink: Stream and batch processing in a single engine. Data Engineering p. 28 (2015)
5. Chen, Y., Ganapathi, A., Katz, R.H.: To compress or not to compress-compute vs. io tradeoffs for mapreduce energy efficiency. In: ACM SIGCOMM workshop on Green networking, New Delhi, India. pp. 23–28 (2010)
6. Chintapalli, S., Dagit, D., Evans, B., Farivar, R., Graves, T., Holderbaugh, M., Liu, Z., Nusbaum, K., Patil, K., Peng, B.J., et al.: Benchmarking streaming computation engines: Storm, flink and spark streaming. In: Parallel and Distributed Processing Symposium Workshops, Chicago, IL, USA. pp. 1789–1792 (2016)
7. Das, T., Zhong, Y., Stoica, I., Shenker, S.: Adaptive stream processing using dynamic batch sizing. In: SoCC, Seattle, WA, USA. pp. 1–13 (2014)
8. Eberle, J., Wijaya, T.K., Aberer, K.: Online unsupervised state recognition in sensor data. In: PerCom, St. Louis, MO, USA. pp. 29–36 (2015)
9. 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)
10. Hu, L., Schwan, K., Amur, H., Chen, X.: Elf: efficient lightweight fast stream processing at scale. In: Usenix ATC, Philadelphia, PA, USA. pp. 25–36 (2014)
11. Liu, M., Ray, M., Zhang, D., Rundensteiner, E.A., Dougherty, D.J., Gupta, C., Wang, S., Ari, I.: Realtime Healthcare Services Via Nested Complex Event Processing Technology. EDBT, Berlin, Germany pp. 622–625 (2012)
12. Lux, M., Chatzichristofis, S.A.: Lire: lucene image retrieval: an extensible java cbir library. In: ACM International conference on Multimedia, Vancouver, British Columbia, Canada. pp. 1085–1088 (2008)
13. LZ4: `https://github.com/jpountz/lz4-java`
14. Nathan Marz's Storm: `https://github.com/nathanmarz/storm`
15. Snappy: `https://github.com/xerial/snappy-java`
16. Tatbul, N., Çetintemel, U., Zdonik, S.: Staying fit: Efficient load shedding techniques for distributed stream processing. In: VLDB. pp. 159–170 (2007)
17. Venkataraman, S., Yang, Z., Franklin, M., Recht, B., Stoica, I.: Ernest: efficient performance prediction for large-scale advanced analytics. In: NSDI, Santa Clara, CA, USA. pp. 363–378 (2016)
18. Yang, F., Tschetter, E., Léauté, X., Ray, N., Merlino, G., Ganguli, D.: Druid: a real-time analytical data store. In: SIGMOD, Snowbird, UT, USA. pp. 157–168 (2014)
19. Zacheilas, N., Kalogeraki, V., Zygouras, N., Panagiotou, N., Gunopulos, D.: Elastic Complex Event Processing exploiting Prediction. Big Data, Santa Clara, CA, USA pp. 213–222 (2015)
20. Zacheilas, N., Zygouras, N., Panagiotou, N., Kalogeraki, V., Gunopulos, D.: Dynamic load balancing techniques for distributed complex event processing systems. In: DAIS, Heraklion, Crete, Greece. pp. 174–188 (2016)
21. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized Streams: Fault Tolerant Streaming Computation at Scale. SOSP, Farmington, PA, USA pp. 423–438 (2013)
22. Zip: `https://docs.oracle.com/javase/7/docs/api/java/util/zip/package-summary.html`