Latency Aware Elastic Switching-based Stream Processing Over Compressed Data Streams

Sajith Ravindra[1], Miyuru Dayarathna[1,2], Sanath Jayasena[2]
[1]WSO2, Inc., [2]Dept of Computer Science & Engineering, University of Moratuwa
sajithr@wso2.com, miyurud@wso2.com, sanath@cse.mrt.ac.lk

ABSTRACT

Elastic scaling of event stream processing systems has gained significant attention recently due to the prevalence of cloud computing technologies. We investigate on the complexities associated with elastic scaling of an event processing system in a private/public cloud scenario. We develop an Elastic Switching Mechanism (ESM) which reduces the overall average latency of event processing jobs by significant amount considering the cost of operating the system. ESM is augmented with adaptive compressing of upstream data. The ESM conducts one of the two types of switching where either part of the data is sent to the public cloud (data switching) or a selected query is sent to the public cloud (query switching) based on the characteristics of the query. We model the operation of the ESM as the function of two binary switching functions. We show that our elastic switching mechanism with compression is capable of handling out-of-order events more efficiently compared to techniques which does not involve compression. We used two application benchmarks called EmailProcessor and a Social Networking Benchmark (SNB2016) to conduct multiple experiments to evaluate the effectiveness of our approach. In a single query deployment with EmailProcessor benchmark we observed that our elastic switching mechanism provides 1.24 seconds average latency improvement per processed event which is 16.70% improvement compared to private cloud only deployment. When presented the option of scaling EmailProcessor with four public cloud VMs ESM further reduced the average latency by 37.55% compared to the single public cloud VM. In a multi-query deployment with both EmailProcessor and SNB2016 we obtained a reduction of average latency of both the queries by 39.61 seconds which is a decrease of 7% of overall latency. These performance figures indicate that our elastic switching mechanism with compressed data streams can effectively reduce the average elapsed time of stream processing happening in private/public clouds.

CCS Concepts

•**Information systems** → **Data stream mining;** *Computing platforms;* •**Networks** → **Cloud computing;**

ICPE'17, April 22-26, 2017, L'Aquila, Italy

DOI: http://dx.doi.org/10.1145/3030207.3030227

Keywords

Event-based systems, Elastic data stream processing, cloud computing, compressed event processing, data compression, IaSS, Software Performance Engineering, System sizing and capacity planning

1. INTRODUCTION

Data stream processing is a paradigm where streams of data are processed to extract useful insights of real world events. Multiple different recent applications of stream processing could be found in the areas of health informatics [1], telecommunications [26], electric grids [12], transportation [15][16], etc. A number of stream processing systems have been introduced in recent times to operate in a variety of software/hardware environments. Most of them are intended to operate in single cloud environments while others operate in multiple clouds. WSO2 CEP [28] server is an example stream processing engine which is designed to operate in multiple clouds.

Infrastructure as a Service (IaaS) service model provides physical and virtual hardware which can be provisioned and decommissioned via a self-service interface [9]. This includes resources such as servers, storage, and networking infrastructure. IaaS provides an environment which is very similar to resources which IT departments within organizations handle.

Data stream processing applications which usually get deployed in private clouds (i.e., private compute clusters) often face resource limitation while their operation due to unexpected loads [2][7]. Multiple approaches exist for mitigating resource limitation issues such as elastically scaling into an external cluster, load shedding, approximate query processing, etc. However, most of them have multiple limitations such as not considering the possible optimizations of the use of network resources, low accuracy of the results produced, and the cost associated with such approaches. Efficient use of compression techniques for optimizing the use of network connection between private and public clouds (IaaS) is one such area which has not been investigated in detail yet.

In this paper we discuss about elastic scaling in a private/public cloud scenario with compressed data streams. We design and implement an Elastic Switching Mechanism (ESM) over private/public cloud system. We use data field compression technique on top of this switching mechanism for compressing the data sent from private cloud to public cloud. We discuss the importance of data switching (sending part of data to public cloud) vs query switching (sending entire query) in the context of ESM. We use two real world data stream processing benchmarks called EmailProcessor and Social Network Benchmark (SNB2016) during the evaluation of the proposed approach. Using multiple experiments on real-world system setup with the two stream processing benchmarks we demon-

strate the effectiveness of our approach for elastic switching-based stream processing using compressed data streams. In a single query deployment with EmailProcessor benchmark we observed that our elastic scaling technique provides 1.24 seconds average latency improvement per processed event which is 16.70% improvement compared to private cloud only deployment. When presented the option of scaling EmailProcessor with four public cloud VMs ESM further reduced the average latency by 37.55% compared to the single public cloud VM. In a multi-query deployment with both EmailProcessor and SNB2016 we obtained a reduction of average latency of both the queries by 39.61 seconds which is a decrease of 7% of overall latency. Specifically the contributions of our work can be listed as follows.

- *Elastic Switching Mechanism (ESM)* - We design and develop a mechanism for conducting elastic scaling of stream processing queries over private/public cloud.
- *Switching at Different Granularities* - We describe means of conducting adaptive switching at different granularities based on the semantics of stream processing queries. Specifically we describe the scenarios of data switching and query switching.
- *Compression with Out-of-order* - We handle out-of-order introduced due to switching with the public cloud. We show our compression technique improved the quality of out-of-order event handling.
- *Evaluation* - We evaluate the proposed approaches by implementing them on real world systems.

The paper is organized as follows. Next, we provide related work in Section 2. An overview for the stream processing software and the benchmarks used in this study are given in Section 3. We provide the details of system design and implementation of the ESM in Section 4. We discuss about the use of data stream compression in ESM in Section 5. Furthermore, we discuss techniques for handling out-of-order introduced by elastic switching in Section 6. The evaluation details are provided in Section 7. We provide the conclusions in Section 8.

2. RELATED WORK

There have been multiple previous work on elastic scaling of event processing systems in cloud environments.

Kleiminger *et al.* studied on implementing a distributed stream processor system based on MapReduce on top of a cloud IaaS to allow it to scale up/down elastically [19]. The main use case of their work was to implement financial algorithms on their framework. They explored how a local stream processor can be deployed in cloud infrastructure to scale to keep up with the expected latency constraints. They mainly talk about two load balancing strategies to achieve this. First one is always-on (load balanced between local and cloud) and second one is adaptive load balancing (move to cloud when the capacity is not enough in local node).

Cloud computing allows for realizing an elastic stream computing service, by dynamically adjusting used resources to the current conditions. Waldemar *et al.* discussed how elastic computing of data streams can be achieved on top of Cloud computing [14]. Features particularly interesting for elastic stream processing in the Cloud includes handling of stream imperfections, guaranteed data safety and delivery, and automatic partitioning and scaling of applications. Load shedding is a well-studied mechanism for reducing the system load by dropping certain events from the stream. Deferred processing of data that cannot be immediately handled are stored for later processing. The assumption of deferred processing is that the overload is limited in time. They mentioned that the most obvious form of elasticity is to scale with the input data rate and the complexity of operations (acquiring new resources when needed, and releasing resources when possible). However, most operators in stream computing are stateful and cannot be easily split up or migrated (e.g., window queries need to store the past sequence of events). In ESM we handle this type of queries by query switching.

Stormy is a system developed to evaluate the "stream processing as service" concept [21]. The idea was to build a distributed stream processing service using techniques used in cloud data storage systems. Stormy is built with scalability, elasticity and multi-tenancy in mind to fit in the cloud environment. They have used distributed hash tables (DHT) to build their solution. They have used DHTs to distribute the queries among multiple nodes and to route events from one query to another. Stormy intent to build a public streaming service where users can add new streams on demand; that is, register and share queries, and instantly run their stream for as long as they want. One of the main limitations in Stormy is it assumes that a query can be completely executed on one node. Stormy is unable to deal with streams for which the incoming event rate exceeds the capacity of a node which is an issue which we address in our work.

Cervino *et al.* tries to solve the problem of providing a resource provisioning mechanism to overcome inherent deficiencies of cloud infrastructure [2]. They have conducted some experiments on Amazon EC2 to investigate the problems that might affect badly on a stream processing system. They have come up with an algorithm to scale up/down the number of VMs(or EC2 instances) based solely on input stream rate. The goal is to keep the system with a given latency and throughput for varying loads by adaptively provisioning VMs for streaming system to scale up/down. In contrast to [21] Cervino *et al.*'s work is focused on running a big query efficiently which is decomposed to smaller quarries which can run on different VMs. However, none of the above mentioned works have investigated on reducing the amount of data sent to public clouds in such elastic scheduling scenarios. In this work we address this issue.

Data stream compression has been studied in the field of data mining. Cuzzocrea *et al.* has conducted a research on a lossy compression method for efficient OLAP [4] over data streams. Their compression method exploits semantics of the reference application and drives the compression process by means of the "degree of interestingness". The goal of this work was to develop a methodology and required data structures to enable summarization of the incoming data stream in order to finally make the usage of advanced analysis/mining tools over data streams more effective and efficient. However, the proposed methodology trades off accuracy and precision for the reduced size.

Jeffery *et al.* has tried to address shortcomings of RFID (Radio-frequency identification) data streams by cleaning the data streams using smoothing filters, they have proposed a middleware layer between the sensors and the application which process data streams. This middleware is responsible for making physical device issues transparent to the higher level application by correcting them at the middleware. The layer/middleware is referred to as "Metaphysical data independenc"(MDI) in their work [18].

Work done in [18] is utilized in [8] to clean and compress data generated by RFID tags deployed in a book store. In this work the data stream is compressed by removing redundant data. They have taken application and deployment semantics into account to develop an efficient data compression method for that specific domain. The MDI layer presented in [18] is customized by adding application specific compression algorithms and they claim that

results in better performance than employing the generic method suggested in [18]. Yanming Nie *et al.* have done a research on inference and compression over RFID data streams [3][24]. They have developed online compression method. The compression of the data stream is mainly achieved by identifying and discarding redundant data.

Multiple work have recently been conducted on privacy preserving data stream mining. Privacy of patient health information has been serious issue in recent times [25]. Fully Homomorphic Encryption (FHE) has been introduced as a solution [10]. FHE is an advanced encryption techniques that allows data to be stored and processed in encrypted form. This provides cloud service providers the opportunity for hosting and processing data without even knowing what the data is. However, current FHE techniques are computationally expensive needing excessive space for keys and cypher texts. However, it has been shown with some experiments done with HElib [11] (an FHE library) that it is practical to implement some basic applications such as streaming sensor data to the cloud and comparing the values to a threshold. We plan to extend this work to the domain of FHE in future.

3. OVERVIEW OF STREAM PROCESSING SOFTWARE

In this section we provide brief description of WSO2 CEP which is the stream processing engine used for implementing the elastic switching mechanism. Furthermore, we provide a short introductions to the EmailProcessor and SNB2016 benchmarks used for the experiments.

3.1 Overview of WSO2 CEP

WSO2 Complex Event Processor (WSO2 CEP) is a lightweight, easy-to-use, stream processing engine. It is available as an open source software under the Apache Software License v2.0 [28]. WSO2 CEP lets users provide queries using an SQL like query language in order to get notifications on interesting realtime events, where it will listen to incoming data streams and generate new events when the conditions given in those queries are met by correlating the incoming data streams.

WSO2 CEP uses a SQL like Event Query language to describe queries. For example, the following query detects the number of taxis dropped-off in each cell in the last 15 minutes [16].

```
from Trip#window.time(15 min)
select count(medallion) as count
group by cellId
insert into OutputStream
```

Listing 1: Example CEP query.

When WSO2 CEP receives a query, it builds a graph of processors to represent the query where each processor is an operator like filter, join, pattern, etc. Input events are injected to the graph, where they propagate through the graph and generate results at the leaf nodes. Processing can be done using a single thread or using multiple threads, where in the latter case we use LMAX Disruptor [27] to exchange events between threads. More details of the WSO2 CEP is available from [28].

3.2 EmailProcessor Benchmark

EmailProcessor is an application benchmark originally designed by Nabi *et al.* [23]. The benchmark is designed around the canonical Enron email data set which is described in [20]. The data set consisted of 517,417 emails with a mean body size of 1.8KB, the largest being 1.92MB. The dataset we used had undergone an offline cleaning and staging phase where all the emails were serialized and stored within a single file with the help of Apache Avro. In our benchmark implementation[1] the data injector read emails from the Avro file, deserialized them and sent to Q1 for filtering. Q1 dropped emails that did not originate from an address ending with `@enron.com`. Furthermore, it removed all email addresses that did not end with `@enron.com` from To, CC, and BCC fields. Q2 modified each and every email by obfuscating the names of three individuals:Kenneth Lay, Jeffrey Skilling, and Andrew Fastow in the email body by replacing their names with Person1, Person2, and Person3 respectively. Q3 operator gathered metrics of each email such as number of characters, words, and paragraphs in the email body. This information is sent to Q5 which aggregated such metrics from multiple Q3 operators in a running window. The processed emails were sent from Q3 to Q4, which were compressed in Avro format and written to `/dev/null` which effectively get discarded. The correct benchmark implementation filters out 89,230 unwanted emails and outputs 42,8187 emails. The architecture of the EmailProcessor benchmark is shown in Figure 1. The choice of EmailProcessor for the experiments was mainly because it does not involve any shared state and it introduced the workload spikes we wanted during the experiments providing good example of real world work load.

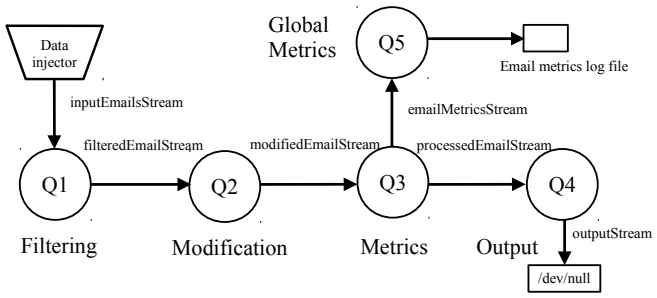

Figure 1: Email Processor

3.3 Social Network Benchmark (SNB2016)

The second application benchmark we used is a version of the DEBS 2016 Grand Challenge which we name as SNB2016 [17] (See Figure 2). SNB2016 is focused on conducting real time streaming analytics on an evolving social network graph. There are four different input streams for SNB2016 Posts, Comments, Friendships, and Likes. The modified version of the benchmark [2] we used in this paper conducts identification of the posts that currently trigger the most activity.

Figure 2: Social Network Benchmark 2016 (SNB2016)

The benchmark calculates the top-3 scoring active posts. An active post P's total score is computed as the sum of its own score and

[1]https://github.com/miyurud/EmailProcessor_Siddhi
[2]https://github.com/miyurud/debs2016

the score of all of its related comments. New posts and new comments each has an initial score of 10 which decreases by 1 each time another 24 hours passes since the post's/comment's creation. The data set used in our experiments consisted of 8,585,497 posts with an average event size of 0.018 Bytes while there were 24,485,315 comments each of 82.09 Bytes. The application logic involved in calculating the top-3 active posts is described in Algorithm 1.

Algorithm 1 Ranker: Find Top 3 Posts

```
1: postMap ← {}
2: commentPostMap ← {}
3: timeWindow ← {}
4: postScoreMap ← {}
5: for all event in stream do
6:     if event.isPost then
7:         postMap.add(event, postScoreMap)
8:         timeWindow.update(event.time, postScoreMap)
9:         timeWindow.addPost(event, postScoreMap)
10:        if postScoreMap.topThreeChanged then
11:            print(postScoreMap.topThree)
12:        end if
13:    else
14:        if event.isCommentToPost then
15:            commentPostMap.update(event)
16:        else
17:            commentPostMap.getParent(event).update()
18:        end if
19:        timeWindow.update(event.time, postScoreMap)
20:        timeWindow.addComment(event, postScoreMap)
21:        if postScoreMap.topThreeChanged then
22:            print(postScoreMap.topThree)
23:        end if
24:    end if
25: end for
```

As shown in Algorithm 1, the Ranker component of the SNB2016 involves accessing a shared time window as well as accessing a shared map data structure called *commentPostMap*. This is an example scenario for a stream processing application which involves shared state. It requires all the events to be sent via the Ranker component.

4. ELASTIC SWITCHING MECHANISM

In this work we present a mechanism for latency aware elastic scaling in private/public cloud scenario. The proposed approach has been implemented on a mechanism called Elastic Switching Mechanism (ESM). The ESM system architecture and two of its application scenarios are shown in Figures 3 and 5.

The ESM is designed to operate between the boundaries of the private and the public clouds. Copies of the same stream processing engine is run in the public cloud when needed. Profiler component gathers the system performance statistics. Scheduler implements the elastic scheduling functions based on the performance information provided by the Receiver. The OOH (Out-of-order handler) component does the reordering when Out-of-order handling has been enforced for a particular stream application. The operations of the ESM is dependent on the Quality of Service (QoS) specification provided by the user.

In our current implementation we use a predefined average latency value as the QoS parameter. Latency is the time spent by an event within the data stream processing system. We use average latency per event as the performance metric since additional latency gets introduced in the case of switching to public cloud. Furthermore, we use the average value rather than 95th percentile value because we want to account for the entire data set sent to public cloud. We had measured 95th percentile average latencies in several performance experiments of ESM and found that 95th latencies obtained from elastic scaling leads to better latency numbers than what is reported in this paper. However, to consider the 5% of the events which are outliers as well as to maintain the simplicity of the presentation we use average latency rather than percentiles. Note that we conduct full processing of stream processing data rather than conducting approximate processing by neglecting high latency events.

Publisher is the component which emits the data stream to public cloud. Data stream compression logic is implemented in the Compresser component. When the data stream sent to the public cloud with compression is enabled it is intercepted by a compression handler component deployed in the public cloud. Compression handler decompresses certain compressed fields as indicated by the preconfigured settings.

The normal mode of operation where the ESM operates within the specified QoS constraints is shown in Figure 3 (a). In this mode the amount of resources in the private cloud is sufficient to maintain user specified QoS. Hence the Stream processing job(s) executes only in the stream processing engine running in the private cloud. Note that at a particular time several stream processing jobs can run in parallel within the stream processing engine (denoted by $Q_1 ... Q_n$ in Figures 3 and 5).

An example for the elastic mode of operation of the ESM is shown in Figure 3 (b). When the scheduler decides that private cloud's resources are insufficient to maintain the QoS specifications, it starts a VM and instantiates a stream processing engine in public cloud. The decision of moving part of the data (data switching) or the entire input data stream (query switching) is dependent on the semantics of the stream processing application. It is indicated to the scheduler via a preconfigured settings. The scheduler starts/migrates the stream processing job which needs to be run in the public cloud based on this preconfigured settings.

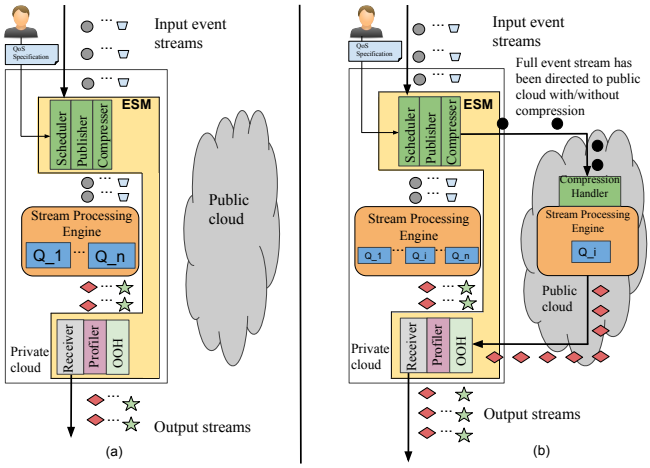

Figure 3: The proposed approach for elastic compressed complex event processing. System operation with single query switched to public cloud with data switching. (a) The private cloud only mode of operation. (b) The hybrid cloud mode of operation with data switching and compression.

The ESM ranks the stream processing jobs/data to be moved to the public cloud based on the difference of their current average

latency and the user specified QoS. The ESM switches these jobs one after the other based on the assigned rank until the results of such switching provides QoS gains. Note that the Figures 3 and 5 indicate different input streams and output streams using different symbols for the sake of clarity of the presentation. Furthermore, the notation used in our paper is explained in Table 1.

4.1 Switching Functions

During the operation of the ESM there are three key switching decisions to be made. First, the public cloud needs to be initialized. The required number of VMs needs to be started and the corresponding stream processing engine needs to be instantiated. In our approach the decision of starting VMs in public cloud in time period t is made considering the average latency threshold observed at the Publisher in the previous time slot $t-1$ (i.e., $L_{t-1} > L_s$). Instantiating VMs is the costliest decision to be made during the system's operation. If N number of VMs each having cost C billed hourly basis are instantiated, irrespective of whether VMs are used for 10 minutes or 50 minutes, a total cost of NC will be incurred. Hence in ESM we start VMs one after the other systematically in such a way that the total cost of VMs can be kept at minimum.

Second, once the VMs get instantiated in the public cloud the decision of when to switch the incoming data stream and which portion to switch is another important decision. As mentioned previously which portion of the data to switch only matters in the case of data switching. In such occasion ESM starts sending data to public cloud when the average latency of the output stream exceeds L_d. When the average latency plummets less than L_d we keep sending data only to private cloud. It should be noted that we maintain the invariant $L_s > L_d$ since L_s is the threshold which makes the costliest operation. After making the decision at the L_s latency level, since the VM is already running in public cloud, sending data to public cloud can be conducted even at a lower latency such as L_d since sending data does not incur a significant cost as with the switching decision taken at L_s. The elastic switching process of data switching of a single stream processing job can be expressed as two binary functions $\phi_{VM}(t)$ and $\phi_{data}(t)$,

$$\phi_{VM}(t) = \begin{cases} 1, L_{t-1} \geq L_s, \tau \; has \; elapsed. \\ \\ 0, D_{t-1} < D_s, L_{t-1} < L_p \qquad \text{Otherwise,} \end{cases}, \tag{1}$$

$$\phi_{data}(t) = \begin{cases} 1, \phi_{VM}(t-1) = 1, L_{t-1} \geq L_d, L_s > L_d \\ \\ 0, \qquad \text{Otherwise,} \end{cases}, \tag{2}$$

where ϕ_{VM} is the function for a single VM and ϕ_{data} is the function for transferring data from private cloud to public cloud. t is the time period for which the values of the binary functions needs to be calculated. A time period of τ has to be elapsed in order for the VM startup process to trigger. D_s is the threshold for total amount of data received by the VM from private cloud. Once the data switching process starts, the QoS value of the stream processing job is measured via latency of the private cloud. The same binary switching function ϕ_{VM} is used for spawning multiple other VMs in the public cloud until the desired QoS latency (L_{QoS}) is achieved for that particular stream processing job.

Finally, once the VMs are instantiated, they need to be checked whether to continue or not once their rental period ends. In the above example, if the VMs are not shutdown before the one hour period elapses another NC amount will be billed by the public cloud. On the otherhand if the need of starting the VM in the public cloud arises just after the VM being shutdown it also incurs considerable time and cost to bring the VM back online. Hence deciding when to shutdown is as important as starting a VM. In our elastic stream processing model we make the decision of when not to continue use of a VM by tracking the amount of data received by the VM ($D_{t-1} = D_s$) during the last operating window $t-1$ and the average latency value recorded by the Publisher running in the private cloud (L_p) during $t-1$. At least D_s amount of data has to be sent to public cloud in order to keep the VM for the next billing session without shutting it down. When the billing period reaches, the switching model evaluates if still the latency is higher than L_p. If the current latency is less than L_p then switching model checks if the number of events sent to public cloud is greater than D_s. The L_p and D_s parameters help the switching model to shutdown underutilized VMs and to retain well utilized VM for next session.

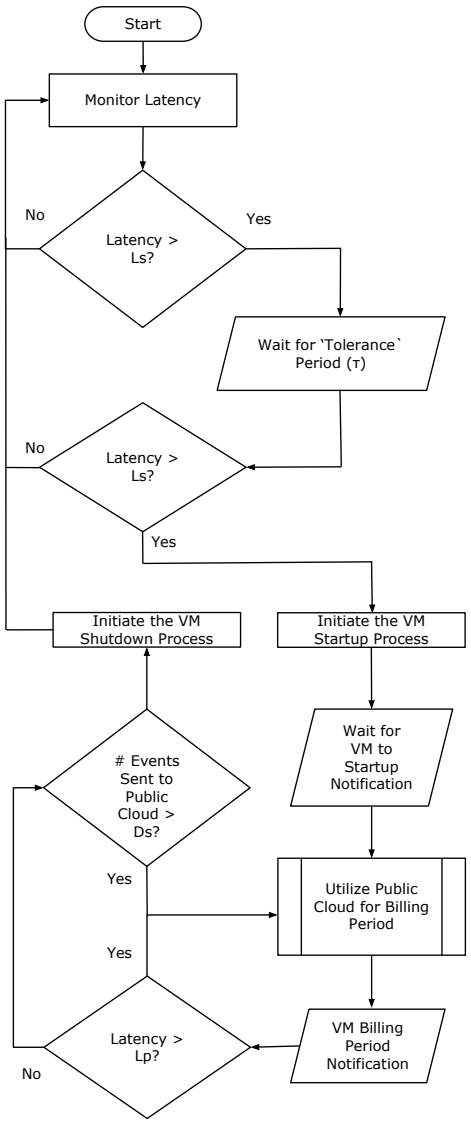

Figure 4: State transition diagram depicting the process of operating a single VM in the public cloud.

The state-transition diagram in Figure 4 describes the process of

Table 1: Notation.

Notation	Description
t	Unit time slot
L_t	Average latency measured at the Receiver component of the ESM during the time slot t
L_s	VM Startup threshold latency. When the average latency exceeds this value the ESM decides to initiate the VM start up process.
L_d	Data switching threshold latency. When $L > L_d$, the ESM starts sending data to public cloud.
τ	Tolerance period. After the unit timeslot (t) elapses, the ESM waits additional τ period before it initiates the VM startup process. In the current implementation of the ESM τ is set equal to t.
L_p	Private cloud threshold latency. At least L_p amount of latency needs to be present in the private cloud for a VM to be kept running in the next unit time slot.
D_t	Total amount of data received by the VM from private cloud during the time slot t
D_s	Threshold for total amount of data received by the VM from private cloud during the time slot t
L_{QoS}	User specified QoS Latency for a single specific stream job.

operating a VM in the public cloud. The rectangle marked as "Utilize Public Cloud for Billing period" indicates the entire process of data/query switching happening in that time frame.

4.2 Data Switching vs Query Switching

An important aspect of our elastic switching mechanism is deciding what type of switching to be conducted when the binary switching function triggers. There are mainly two types of switching as data switching and query switching.

In data switching a portion of the data is switched to the public cloud. This is feasible only if the queries used in stream processing application allows for split-merge (fission) of data items [6]. If the query operators need to access a shared state (i.e., a shared memory location) such data switching is not possible. The EmailProcessor benchmark is an example for such stream processing application which allows for fission. Elastic switching into the public cloud introduces a significant latency to the events sent to the public cloud. Hence in the context of data switching we send only R percentage of the input data stream to public cloud to avoid unnecessary latency overheads. In the experiments conducted in this paper we selected to use $R = 10$ after conducting multiple experiments with different R values with all the other parameters fixed. For example, we run the same experiment multiple times in each round with different R values such as 1,2,4,8,10,12,14,20, etc.

On the otherhand there are certain queries which require the entire set of input events to be passed through the operators to produce correct results (i.e., need to access shared state). For example, for the correct operation of SNB2016 benchmark, the entire event stream needs to be passed through the query operators of its Ranker component. Aggregation of results is another example. Hence in such situations data switching cannot be performed. Instead the whole stream processing application needs to be migrated to the public cloud. As indicated by the bent arrow, Q_i has been completely transferred from the private cloud to public cloud.

5. EVENT STREAM COMPRESSION

Several key techniques exist for compressing event streams before they are sent to the public cloud. First technique is Compression by Field Trimming (i.e., query parameter tuning) where the input queries are adjusted and tuned in such away that only the required set of fields are sent to the public cloud. The second technique is by field compression where large data items of an event gets compressed using a compression algorithm. In this paper we implement only the former technique while we discuss the field compression for the sake of completeness of the discussion.

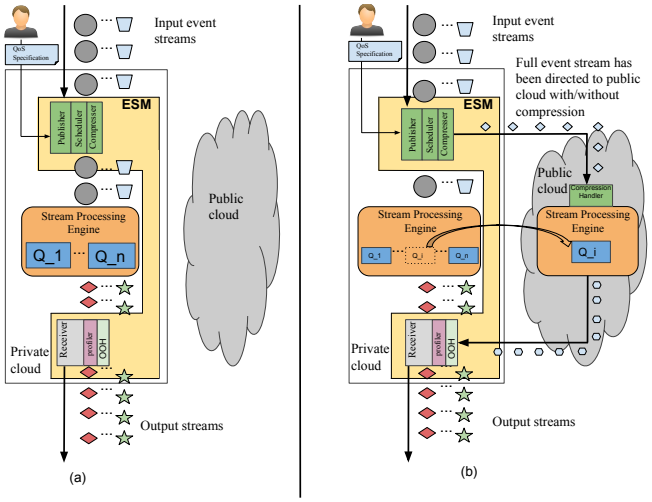

Figure 5: System operation with single query switched to public cloud with query switching. (a) The private cloud only mode of operation. (b) The hybrid cloud mode of operation with query switching without compression.

5.1 Compression by Field Trimming

Query transformation is a technique which converts a given stream processing query into another form which effectively gives the same results when executed by the event processor engine [6].

The query transformation technique involves static analysis of the query and deciding which fields can be omitted by looking at the query. If a certain field is not used in the query we can drop that particular event from the data stream, and thereby reduce the amount of data sent through the wire. For example, lets consider the example in Figure 6. In the query shown on top only 3 fields out of 8 fields available in the data stream is included in the output. Therefore, by looking at the output it can be concluded it is safe to omit sending non-used fields that are not in the output. In this particular example we can omit sending iij_timestamp, fromAddress, bccAddresses, subject, and regexstr fields.

5.2 Data Field Compression

In the data field compression technique, the contents of preselected set of data fields are compressed using Gzip compression algorithm. The selection of the fields to be compressed is done via offline profiling. Once the fields to be compressed are selected,

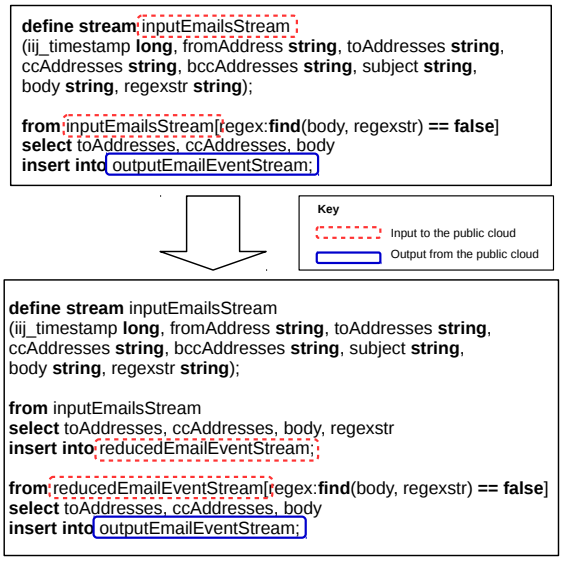

Figure 6: An example of query transformation based event stream compression.

they are configured in the elastic switching mechanism through a configuration file. While one could argue that involvement of an additional step of compression may increase average latency per event, our experiments in real world environment has shown this is not the case.

6. OUT-OF-ORDER EVENTS HANDLING

Certain stream processing applications are sensitive for out-of-order events [22]. There are two ways in which out-of-order events could get introduced to the output stream from elastic stream processing system. First, due to the switching activity with public cloud which is unavoidable in an stream processing system. Second, out-of-order events can already be present in the input stream for the stream processing system. From multiple experiments conducted on the above two scenarios we observed that the out-of-ordering introduced by switching into public cloud far outperforms the out-of-ordering which could occur naturally in the input stream. For example, the effect of out-of-ordering introduced by the input stream to become a significant factor compared to the out-of-order introduced due to switching to the public cloud, we found that at least about 60% of the events in the input stream needs to be arriving out-of-order. This is uncommon in many stream processing scenarios. Hence in this paper we ignore the out-of-order which is naturally present in the input streams.

There are four main techniques of disorder handling: Buffer-based, Punctuation-based, Speculation-based, and Approximation-based. In this paper we discuss the use of a buffer for reordering out-of-order events which are gathered from both the private and public clouds. We allow user to specify as a parameter whether the event stream output from the stream processing system needs to handle out-of-order events or not. We use buffer-based technique called reordering which essentially reorders events at the output and periodically flushes the reordered events to the output stream.

We implement out-of-order event handling at the receiver of the stream processing system. We detect whether there are any out-of-order event in the input stream. If found we send those out-of-order events to private cloud to avoid further delays of processing the events. We have found that this technique gives higher quality output compared to a scenario of buffer-based reordering which does not send events to public cloud at all.

7. EVALUATION

The evaluations described in this paper were conducted on 2 physical computers and on 4 Virtual Machines deployed in a compute cluster of WSO2. Two physical computers were used in all the experiments described in Sections 7.1 to 7.4. One of the computers was used as the public cloud and the other computer as the private cloud. The public cloud computer was running on Intel® Core i5® M560, 2.67 GHz. Each VM had 4 cores, 3072KB L2 cache and 32KB L1(d/i) caches. The public cloud computer had 4GB RAM. The private cloud computer was a 64-bit Genuine Intel® Core™ i7-4800MQ CPU which operates at 2.70GHz, 8 cores, 8 hardware threads. The sizes of the L1(d/i), L2, and L3 caches of this computer were 32KB, 256KB, and 6144KB respectively. We used Oracle JDK 1.8.0_101 and WSO2 CEP Server 4.0.0 during the experiments. The private and public clouds were connected via a wireless network. We used NetEm network emulator [13] to simulate 70ms latency which is present between two data centers operating in the East and the West coasts of the United States [5].

Each VM out of the 4 VMs were running 64-bit Intel® Xeon E312xx (Sandy Bridge) which operate at 2.70GHz. Each VM had one CPU socket, with 2 cores. The sizes of the L1(d/i), L2, and L3 caches were 32KB, 32KB, and 4096KB respectively. Each VM had 4GB RAM with one 40GB hard disk. Each VM was installed with Linux Ubuntu (kernel 3.13.0-36-generic).

Note that the ESM's source code has been released under open-source license[3][4].

7.1 Elastic Switching Without Event Stream Compression

In the first round of the experiments we evaluated the performance benefits of our elastic scaling mechanism. We used EmailProcessor benchmark for this purpose and one computer was used as the public cloud while the private cloud was running in another computer. We conducted performance experiments to measure how effective would our stream processing system be in reducing the overall average latency of processing P number of input events. We use average latency as the performance metric because in an elastic scaling system event processing latency acts as an indicator of system performance. The results present in this paper are taken single round experiments. The input data rate variation of the EmailProcessor benchmark during the experiments is shown in Figure 7. This indicates that the workload distribution had significant variations. VM rental period was set to 1 minute to reduce the time taken during the experiments while VM setup period was set to 10 seconds. In our current implementation we do initiate the full stream processing job in the public cloud during an elastic scaling operation. The latency thresholds L_s and L_d were set as 12,000 and 10,000 milliseconds respectively.

Figure 8 indicates the latency distribution of running the EmailProcessor benchmark both without switching to public cloud and with switching to public cloud. It can be observed that there are two significant spikes in the workload distribution. Figure 8 shows the average latency variation of the private cloud without switching and with switching scenarios. The four dotted vertical lines on Figure 8 indicates the points in the timeline where VM start/VM

[3]https://github.com/sajithshn/event-publisher
[4]https://github.com/sajithshn/statistics-collector

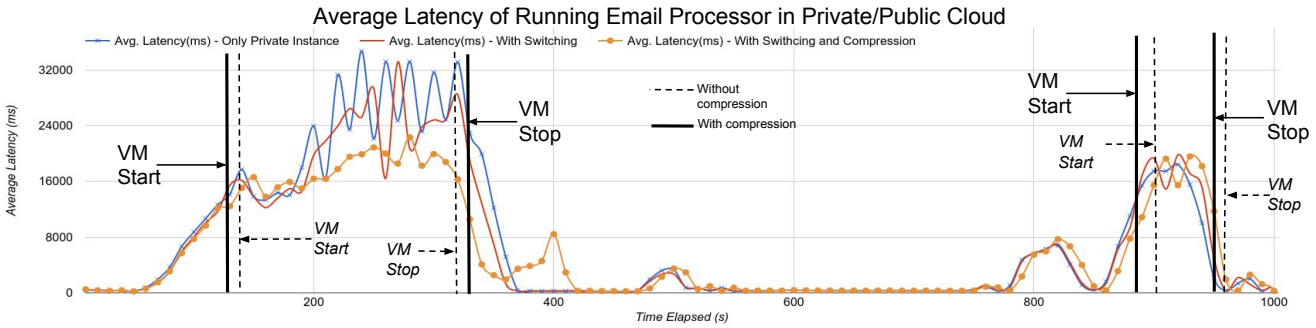

Figure 8: Average Latency of Running EmailProcessor in Private/Public Cloud

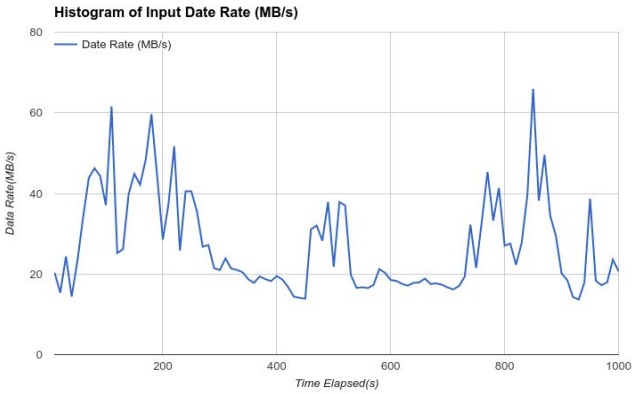

Figure 7: Input data rate variation

shutdown operations has been initiated. Elastic scaling system instantiates the VM in the public cloud at 130^{th} second. When wall clock is 340 seconds, the VM is shutdown. The benchmark is run only in the private cloud in the time range between 340 seconds and 890 seconds. At 890 seconds the VM instantiation process starts again. The VM shutsdown when wall clock time is 950 seconds. The results give notable results with an improvement of the average latency of 0.581 seconds in hybrid cloud (i.e., with switching) which is 7.84% improvement compared to private cloud only deployment. The 7.84% improvement of the average latency was calculated by taking the average values of the data points corresponding to the "Avg. Latency(ms) - Only Private Instance" and "Avg. Latency(ms) - With Switching" curves. The difference of the two averages was 0.581 seconds which was 7.84% improvement compared to "Avg. Latency(ms) - Only Private Instance" case. Note that we do not plot when actually the data stream gets switched to/from the public cloud to maintain the simplicity of the charts. Furthermore, the elastic switching reduces the load average of the system operation. This can be observed from Figure 8.

7.2 Elastic Switching with Event Stream Compression

Next, we applied event stream compression on top of the elastic switching operations of the ESM. The results are shown in Figure 8. The four vertical continuous lines indicates where the VM start/shutdown has been initiated with this round of experiments. It can be observed a significant latency gain has been made with use of event stream compression on the events sent to the public cloud. ESM instantiates the VM in the public cloud at 140^{th} second. The VM is shutdown when the wall clock was 330 seconds. Again the VM was started at 900th second and was shutdown at 960th second. The results obtained from event stream compression was impressive compared to the naive private cloud only operation. ESM provided 1.24 seconds per event average latency gain compared to the private cloud only operation. This is a 16.70% latency improvement. Furthermore, event stream compression provides us 8.86% more performance gain compared to elastic switching which does not involve compression.

7.3 Elastic Switching in the Presence of Multiple Queries

Next, we run multiple queries simultaneously in the private cloud and elastically scale one of the queries to the public cloud when the load in the private cloud exceeds specified QoS constraints. Another objectives of this experiment was to investigate on the performance variation happen when query switching has been conducted with the ESM.

Figure 9 (a) indicates how average latency variation occurs in both EmailProcessor and SNB2016 benchmarks when deployed only in the private cloud. Figure 9 (b) shows how latency variation happens when only SNB2016 has been deployed in the public cloud. Note that in this case we completely take SNB2016 out from private cloud and deploy it in the public cloud while EmailProcessor continues to run as it is in the private cloud. Similar to the previous experiment we have marked the switching points in Figure 9 (b). In this experiment ESM has started the VM in the public cloud at 220^{th} second while the VM was shutdown at 340^{th} second. Again the VM has been started at 960^{th} second and shutdown at 980^{th} second.

It can be observed that our elastic scaling mechanism successfully reduces the average latency of the SNB2016. Although migration of the SNB2016 to public cloud completely has resulted in slight increase of average latency of SNB2016 by 34.89 seconds (7.58% increase), the performance benefits of elastic scaling far outweighs that drawback. Specifically the elastic scaling of SNB2016 reduces the average latency of EmailProcessor by 74.47 seconds which is 15.75% improvement in average latency of the EmailProcessor. Overall the elastic scaling reduces the average latency of both the queries by 39.61 seconds which is a decrease of the overall average latency by 7%. Note that we do not investigate on load average variation in this experiment because we have only one query compared to the non-switching scenario.

7.4 Effect of Elastic Switching with Reordering

Elastic scaling into a public cloud in general introduces disor-

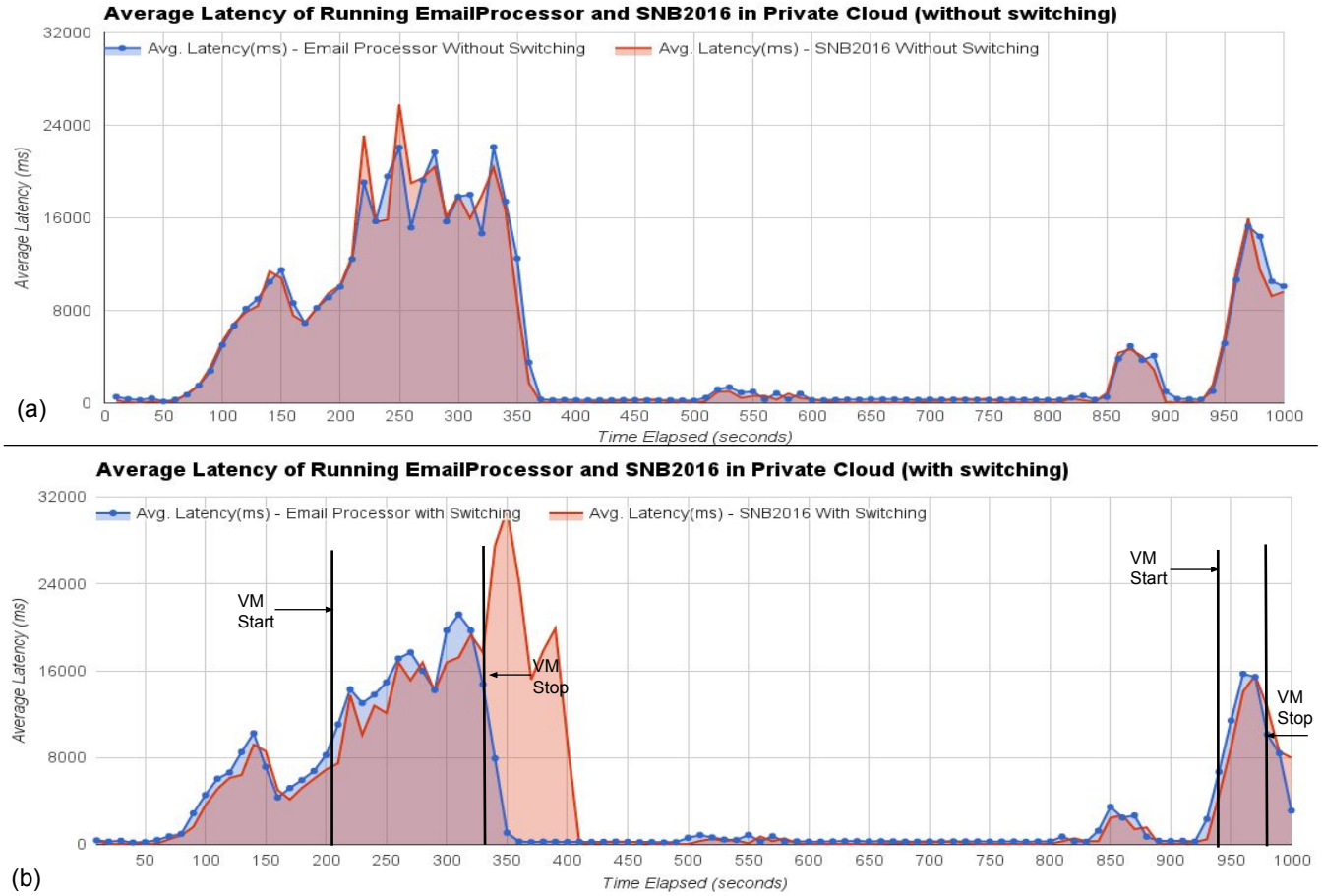

Figure 9: Evaluation of the effectiveness of the ESM in the presence of multiple queries. (a) Latency variation for EmailProcessor and SNB2016 where both benchmarks run on private cloud. (b) Latency variation for hybrid cloud operation where SNB2016 getting deployed in the public cloud.

der to an event stream. In this experiment we use a buffer in the Receiver component of ESM to gather events sent from both public and private clouds and sort them. A timer is used to flush events periodically. The results of running EmailProcessor benchmark with data switching with/without compression is shown in Figure 10.

Figure 10: Effect of reordering on the average latency of the event streams output by ESM.

The experiment results on Figure 10 indicated that buffer-based reordering with compression provides 16 seconds average latency improvement compared to conducting the same reordering on a scenario without compression. Use of data stream compression introduced 19.44% latency improvement when compression has been used. Furthermore, we observed that out-of-order events percentage at the Receiver was 45.40% when reordering was conducted on elastic data stream without compression. However, reordering on the elastic data stream with compression reduced the percentage of out-of-order events further by 7% to 38.4%. These results indicates that our data stream compression based ESM can better handle out-of-order events which are produced in general on naive elastic scaling mechanisms.

7.5 Elastic Switching with Multiple Public Cloud VMs

Experiments described till this point were conducted only using single VM instance in public cloud. In this experiment we demonstrate the ability of conducting elastic scaling on multiple VMs. Each VM was configured to run on separate computer. The scaling experiments were conducted with maximum four VMs on four physical computers running in the public cloud. We observed that although we added more VMs (and physical computers) to the public cloud the switching mechanism did not start more VMs other than four due to the system conditions did not satisfy the conditions of the binary switching function $\phi_{VM}(t)$. The results of this experiment are shown in Figure 11.

It can be observed from Figure 11 that overall average latency reduces significantly when two public cloud VMs were used than sin-

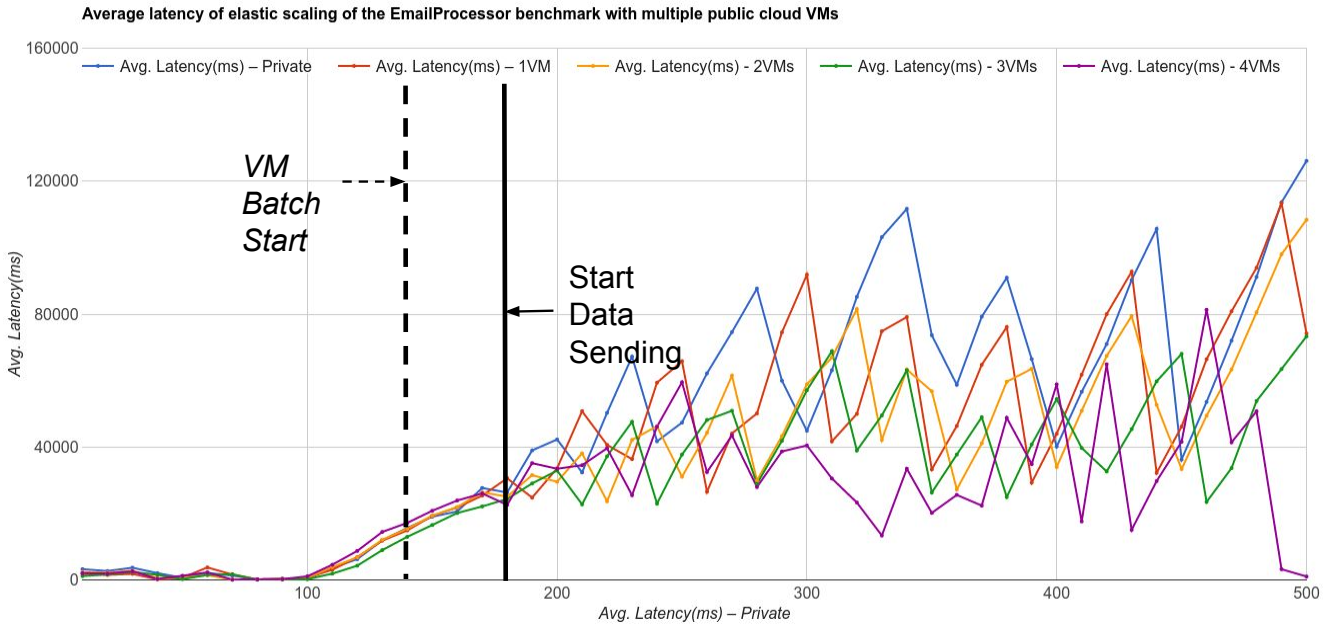

Figure 11: Average latency of elastic scaling of the EmailProcessor benchmark with multiple public cloud VMs

gle VM. The naive private cloud only average latency was 129.32 seconds. When a single VM on public cloud was started by the ESM the per event latency was further reduced by 53.33 seconds. This was a 15.1% average latency improvement compared to private cloud only deployment. However, when the ESM was configured to use maximum 2 VMs the overall average latency was reduced to 110.78 seconds which is a reduction of 18.54 seconds (22.8%) compared to the private cloud only deployment. Introduction of 3rd and 4th public cloud VMs resulted in further reductions in average latency by 36% and 47% compared to the private cloud only mode. Compared to the single public cloud VM, the 4 public cloud VMs further reduce the latency by 37.55%. These results indicate that ESM is capable of elastically scaling to multiple public cloud VMs and Scaling introduces significant performance gains. Note that the results shown on Figure 11 does not include the scenarios of VM shutdown since the motivation of the experiment was to indicate that the proposed ESM can reduce the overall latency further to considerable extent when additional VMs were added in the public cloud.

7.6 Discussion

The above five different experiments indicate the effectiveness of our elastic stream processing mechanism in lowering the average latency of data stream processing. In this paper our focus was on stream processing jobs which get deployed in public cloud completely during such elastic scaling scenario. Elastic scaling of a portion of the stream processing job is an important area. For example, the last three query operators of the EmailProcessor benchmark (Q_3, Q_4, and Q_5) can be transferred to public cloud while the first two operators (Q_1 and Q_2) could still remain in the private cloud.

We have demonstrated how effective it is to use compression along with elastic switching to handle out-of-order events which gets produced due to the use of public cloud. However, a limitation of our current implementation is that out-of-order event handling implementation could not turn off the VM once the switching to public cloud happens. This is due to the added latency of the buffer based re-ordering. We are working on to fix this issue.

We have used average latency as the user specified QoS metric in the current implementation because latency is one of the most important performance metrics in an elastic stream processing mechanism. However, we plan to investigate on use of other metrics such as throughput, system load average, etc. as parameters for switching process in future. We need to consider higher percentiles when there are a large number of outlier values in a dataset. We see two potential uses of higher percentile latencies in relation to ESM. First, for the switching function where the decision for switching needs to be taken. Because there are less number of events with outlier latencies in the private cloud mode of operation there is no significant difference in the use of average latency or a percentile such as 95th percentile for taking the switching decision. The switching decision is influenced only by the latency of the private cloud. Hence we opt to use average latency to make the switching decision. The second place where higher latencies would matter is measuring the latency benefit of elastic scaling. Switching to public cloud inherently introduces events with outlier latencies to the event stream. However, the benefit of ESM realizes when we consider the overall picture using the average latency. In hybrid cloud (public+private cloud) operation, although there are few outlier events with large latencies, their effect is subdued by the overall average latency reduction achieved by elastic switching. Hence, for measuring the elastic scaling benefit of using ESM we used the reduction of average latency but not the reduction of higher percentiles such as 95th percentile.

It should be noted that the ESM presented in this paper reorders events only at the output of the streaming pipeline. This assumes that the operations conducted within the elastic stream processing pipeline (i.e., the public cloud portion of the stream processing

pipeline) are not sensitive for the order of the events.

We used two application benchmarks which we believe are good example applications for stream processing. However, there are other different types of applications which involve complex operations which we plan to conduct in future. Event pattern matching, event sequence matching, sliding window operations, etc. are some examples for such applications.

8. CONCLUSIONS

In this paper we present an elastic switching mechanism (ESM) for elastic scaling of data stream processing systems. ESM leverages public cloud to augment a private cloud when the private cloud is overloaded. We implement the proposed approach on WSO2 Complex Event Processsor (WSO2 CEP). We have tested the feasibility of our approach by using two application benchmarks (i.e., queries) called EmailProcessor and SNB2016. We observe that in both data switching and query switching elastic scaling scenarios our elastic switching mechanism provides performance benefits in terms of latency reductions. In the case of the data switching scenario with single public cloud VM instance we obtained 16.7% improvement of average latency compared to private cloud only operation. The proposed compressed stream processing approach achieves 8.86% performance gain compared to naive elastic switching. In another experiment which involves query switching we obtained 7% improvement of overall average latency. Furthermore, we demonstrated that our data field compression based ESM could effectively produce lesser out-of-order events by 7% compared to a scheme which does not involve data stream compression. Moreover, when presented the option of scaling EmailProcessor with four public cloud VMs ESM further reduced the average latency by 37.55% compared to the single public cloud VM. These performance figures indicate that our elastic scaling mechanism can effectively reduce the average elapsed time spent on stream data processing.

Currently we do switch entire query to public cloud in the case of query switching. In future we plan to migrate only portions of the queries which will lead for better control of the elastic switching process. We also plan to investigate and implement Fully Homomorphic Encryption based ESM which will be useful for implementing stream processing applications which involve sensitive data such as processing health records. Furthermore, we plan to add microbenchmarks for different components in the ESM as well as use more elaborate performance models which may consider multiple aspects such as immediate saturation of the latencies. While a network emulator has been used to simulate latency between two distant data centers, we plan to conduct experiments in real cloud environments in future.

9. REFERENCES

[1] M. Blount, M. Ebling, J. Eklund, A. James, C. McGregor, N. Percival, K. Smith, and D. Sow. Real-time analysis for intensive care: Development and deployment of the artemis analytic system. *Engineering in Medicine and Biology Magazine, IEEE*, 29(2):110–118, March 2010.

[2] J. Cervino, E. Kalyvianaki, J. Salvachua, and P. Pietzuch. Adaptive provisioning of stream processing systems in the cloud. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 295–301, April 2012.

[3] R. Cocci, T. Tran, Y. Diao, and P. Shenoy. Efficient data interpretation and compression over rfid streams. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1445–1447, April 2008.

[4] A. Cuzzocrea and S. Chakravarthy. Event-based lossy compression for effective and efficient {OLAP} over data streams. *Data & Knowledge Engineering*, 69(7):678 – 708, 2010. Advanced Knowledge-based Systems.

[5] Datapath. Datapath.io. URL: http://console.datapath.io/map, 2016.

[6] M. Dayarathna and T. Suzumura. Automatic optimization of stream programs via source program operator graph transformations. *Distributed and Parallel Databases*, 31(4):543–599, 2013.

[7] M. Dayarathna and T. Suzumura. *A Mechanism for Stream Program Performance Recovery in Resource Limited Compute Clusters*, pages 164–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[8] N. Dindar, Ã. Balkesen, K. Kromwijk, and N. Tatbul. Event processing support for cross-reality environments. *IEEE Pervasive Computing*, 8(3):34–41, July 2009.

[9] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud Computing Fundamentals*, pages 21–78. Springer Vienna, Vienna, 2014.

[10] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[11] S. Halevi and V. Shoup. *Algorithms in HElib*, pages 554–571. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[12] J. Hazra, K. Das, D. P. Seetharam, and A. Singhee. Stream computing based synchrophasor application for power grids. In *Proceedings of the First International Workshop on High Performance Computing, Networking and Analytics for the Power Grid*, HiPCNA-PG '11, pages 43–50, New York, NY, USA, 2011. ACM.

[13] S. Hemminger. Network emulation with netem. 2005.

[14] W. Hummer, B. Satzger, and S. Dustdar. Elastic stream processing in the cloud. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 3(5):333–345, 2013.

[15] T. Hunter, T. Das, M. Zaharia, P. Abbeel, and A. Bayen. Large-scale estimation in cyberphysical systems using streaming data: A case study with arterial traffic estimation. *Automation Science and Engineering, IEEE Transactions on*, 10(4):884–898, Oct 2013.

[16] S. Jayasekara, S. Perera, M. Dayarathna, and S. Suhothayan. Continuous analytics on geospatial data streams with wso2 complex event processor. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 277–284, New York, NY, USA, 2015. ACM.

[17] M. Jayasinghe, A. Jayawardena, B. Rupasinghe, M. Dayarathna, S. Perera, S. Suhothayan, and I. Perera. Continuous analytics on graph data streams using wso2 complex event processor. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 301–308, New York, NY, USA, 2016. ACM.

[18] S. R. Jeffery, M. J. Franklin, and M. Garofalakis. An adaptive rfid middleware for supporting metaphysical data independence. *The VLDB Journal*, 17(2):265–289, 2008.

[19] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch. Balancing load in stream processing with the cloud. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, pages 16–21, April 2011.

[20] B. Klimt and Y. Yang. Introducing the enron corpus. In *CEAS 2004 - First Conference on Email and Anti-Spam, July 30-31, 2004, Mountain View, California, USA*, 2004.

[21] S. Loesing, M. Hentschel, T. Kraska, and D. Kossmann. Stormy: An elastic and highly available streaming service in the cloud. In *Proceedings of the 2012 Joint EDBT/ICDT Workshops*, EDBT-ICDT '12, pages 55–60, New York, NY, USA, 2012. ACM.

[22] C. Mutschler and M. Philippsen. Distributed low-latency out-of-order event processing for high data rate sensor streams. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 1133–1144, May 2013.

[23] Z. Nabi, E. Bouillet, A. Bainbridge, and C. Thomas. Of streams and storms. *IBM White Paper*, 2014.

[24] Y. Nie, R. Cocci, Z. Cao, Y. Diao, and P. Shenoy. Spire: Efficient data inference and compression over rfid streams. *IEEE Transactions on Knowledge and Data Engineering*, 24(1):141–155, Jan 2012.

[25] A. Page, O. Kocabas, S. Ames, M. Venkitasubramaniam, and T. Soyata. Cloud-based secure health monitoring: Optimizing fully-homomorphic encryption for streaming algorithms. In *2014 IEEE Globecom Workshops (GC Wkshps)*, pages 48–52, Dec 2014.

[26] B. Theeten, I. Bedini, P. Cogan, A. Sala, and T. Cucinotta. Towards the optimization of a parallel streaming engine for telco applications. *Bell Labs Technical Journal*, 18(4):181–197, 2014.

[27] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart. High performance alternative to bounded queues for exchanging data between concurrent threads. *technical paper, LMAX Exchange*, 2011.

[28] WSO2. Wso2 complex event processor. URL: http://wso2.com/products/complex-event-processor/, 2016.