Adaptive Provisioning of Stream Processing Systems in the Cloud

Javier Cerviño[#1], Evangelia Kalyvianaki[*2], Joaquín Salvachúa[#3], Peter Pietzuch[*4]

[#]*Dto. Ingeniería de Sistemas Telemáticos, Universidad Politécnica de Madrid*
Avda. Complutense s/n, 28040 Madrid, Spain
[1]`jcervino@dit.upm.es` [3]`jsalvachua@dit.upm.es`

[*]*Department of Computing, Imperial College London*
180 Queen's Gate, South Kensington Campus, London SW7 2AZ, United Kingdom
[2]`ekalyv@doc.ic.ac.uk` [4]`prp@doc.ic.ac.uk`

***Abstract*— With the advent of data-intensive applications that generate large volumes of real-time data, distributed stream processing systems (DSPS) become increasingly important in domains such as social networking and web analytics. In practice, DSPSs must handle highly variable workloads caused by unpredictable changes in stream rates. Cloud computing offers an elastic infrastructure that DSPSs can use to obtain resources on-demand, but an open problem is to decide on the correct resource allocation when deploying DSPSs in the cloud.**

This paper proposes an adaptive approach for provisioning virtual machines (VMs) for the use of a DSPS in the cloud. We initially perform a set of benchmarks across performance metrics such as network latency and jitter to explore the feasibility of cloud-based DSPS deployments. Based on these results, we propose an algorithm for VM provisioning for DSPSs that reacts to changes in the stream workload. Through a prototype implementation on Amazon EC2, we show that our approach can achieve low-latency stream processing when VMs are not overloaded, while adjusting resources dynamically with workload changes.

I. Introduction

There has been a surge in data-oriented applications in a range of domains, including financial processing, web analytics, social networks and healthcare systems, which must process continuous data streams. To this end, *distributed stream processing systems* (DSPSs) execute continuous *queries* in real-time [1], [2]. In DSPSs, data comes from sources and is transformed by processing elements—usually referred to as *operators*—that are interconnected to realise given query semantics. Important performance requirements for continuous queries are low latency and adequate throughput to handle all input data in real-time.

Cloud computing has emerged as a flexible paradigm for facilitating resource management for elastic application deployments at unprecedented scale. Cloud providers offer a shared set of machines to cloud tenants, often following an *Infrastructure-as-a-Service* (IaaS) model. Tenants create their own virtual infrastructures on top of physical resources through virtualisation [3]. *Virtual machines* (VMs) then act as execution environments for applications.

While deployments of DSPSs in the cloud promise to exploit its inherent elasticity, an open problem is how to provision resources (i.e. CPU, memory and network bandwidth) in response to sudden, time-varying changes in stream workloads. While cloud infrastructures are engineered to offer VMs on-demand, this has been applied most successfully to batch processing models such as map/reduce [4]. The real-time nature of stream processing poses unique challenges due to the need to achieve low latency and guaranteed throughput. Our goal is to explore the performance implications of deploying DSPSs on a public cloud infrastructure and to propose an approach for allocating VMs based on workload demands in order to maintain a given target throughput with low latency.

Resource management in DSPSs has been an active research area for many years, with an emphasis on mechanisms for handling overload. Previous solutions such as load-shedding [5], [6], [7] , admission control [8], adaptive query planning [9], load balancing [10], [11] and efficient initial operator placement [12], do not address overload by obtaining additional resources on-demand. In previous work [13], we presented a hybrid approach that balances stream processing between a local processor and cloud resources, but the focus of the work was not on the elastic behaviour within the cloud. In contrast, cloud deployments of DSPSs need mechanisms for making decisions *when* to update their resource allocation in response to workload changes and *how*.

In this paper, we propose an approach for on-demand provisioning of resources in a cloud-deployed DSPS. Through a set of benchmarks, we first explore the suitability of a public cloud infrastructure for stream processing by measuring network and processing latencies, jitter and throughput. Our results show that the main factor dominating performance is the latency introduced by the cloud deployment. On the other hand, there is no significant throughput reduction when VMs are not overloaded. Based on these results, we propose an adaptive algorithm that resizes the number of VMs in a DSPS deployment in response to workload demands by taking throughput measurements of each involved VM. We evaluate our provisioning approach experimentally by deploying it as part of a DSPS on the Amazon Elastic Compute Cloud (EC2) [14]. We show that it works well regardless of the computing power of the underlying VMs.

In the next section, we review previous studies of cloud performance. In §III, we report new benchmarking results for stream processing on the Amazon EC2 cloud by measuring jitter and end-to-end latency. §IV presents a novel adaptive algorithm to dynamically allocate VMs in the presence of workload changes, and we evaluate our approach on Amazon EC2 in §V. Finally we discuss future work in §VI and draw conclusions in §VII.

II. Background

Next we review related work on performance of public clouds for different application domains with the aim of understanding the implications of deploying DSPSs in clouds.

Network performance. Rehr et al. [15] present preliminary network performance measurements running the widely used x-ray spectroscopy benchmark on Amazon EC2. They report that network performance is substantially lower than what can normally be achieved in academic compute clusters. Wang et al. [16] study end-to-end network performance among Amazon EC2 VMs where they observe abnormal delay variations and unstable TCP/UDP throughput. Barker et al. [17] evaluate the effectiveness of cloud platforms for running latency-sensitive multimedia applications. Their results reveal that the experienced latencies and jitter are fairly typical of wide-area networks. Jackson et al. [18] indicate that Amazon EC2 clusters are slower than other typical mid-range Linux clusters or high-performance computing systems and that the network limits performance, causing significant variability in applications. Finally, Evangelinos et al. [19] report that public cloud resources are comparable to low-cost clusters, in which latency and bandwidth are one to two orders of magnitude lower compared to large compute centre facilities.

CPU performance. Ostermann et al. [20] indicate that performance and reliability of Amazon EC2 is insufficient for scientific computing at large. Wang et al. [16] find that co-location of VMs causes decline in overall performance of VMs. Barker et al. [17] show that jitter and throughput measured in an application can degrade when VMs are co-located. Finally, Dittrich et al. [21] compare data from running a map/reduce application on Amazon EC2 for a month with results obtained on a local cluster. They observe highly variable CPU performance across different processor architectures.

Discussion. Although not all studies agree, latency instabilities arise in Amazon EC2 in the face of intensive network usage. In addition, the VMs offered are not fully isolated and hence experience some performance variability when running high-performance compute tasks. An open question is how this may affect the performance of cloud deployments of DSPSs due to time critical nature of stream processing. On the one hand, stream processing may not fully utilise a virtual processor when there is sufficient headroom given particular input data rates. Maintaining headroom is important to avoid unnecessary queueing or discarded data items due to load shedding. On the other hand, stream processing has fixed resource requirements to handle a given stream rate. Variance in the availability of network and processing resources may impact user-observed performance when the minimum resource requirements cannot be achieved during transient periods of time.

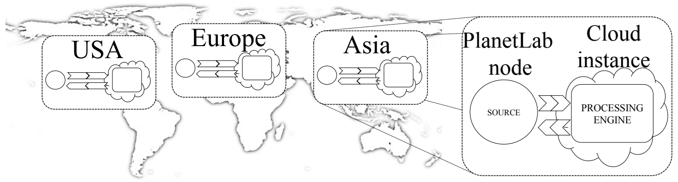


Fig. 1. Experimental set-up for network measurements

III. Cloud Stream Processing Performance

Our initial goal is to answer the following questions: Is it feasible to stream data from sources on the Internet to various public cloud data centre (DC) sites, as provided by Amazon or Google? In other words, can wide-area Internet paths support streaming data into cloud DCs? We also investigate if IaaS clouds are suitable for hosting stream processing systems. Do best effort VMs have a sufficient level of predictability to support low-latency, low-jitter and high-throughput stream processing? Is the computational power of Amazon EC2 VMs appropriate when used for standard stream processes tasks?

A. Network measurements

The network parameters that affect stream processing conditions are *latency*, *jitter* and *bandwidth*. Latency is important because it contributes to the end-to-end delay of a real-time processing system. Jitter matters for several reasons: first, it must be bounded, otherwise it increases overall latency caused by the usage of large buffers; second, it changes tuple inter-arrival times in a DSPS and thus reduces system predictability. Finally, bandwidth restricts achievable processing throughput.

Experimental set-up. We implement a scenario, in which a node (i.e. the *stream source*) sends stream data to another node (i.e. the *stream processing engine*) for processing. We place the engines in different Amazon DCs distributed around the world: we select a DC in the US, one in Europe and a third in Asia (cf. Fig. 1), repeating a set of measurements in each. The engines use a large Amazon EC2 instance, which has 7.5 GB of memory and 4 EC2 compute units[1]. The sources are hosted on nine PlanetLab nodes to emulate a global set of data sources (three in Asia, three in Europe and three in the US). Each set of nodes is associated with the nearest Amazon EC2 DC.

For each DC location, we conduct several experiments divided into three basic runs, involving a different PlanetLab stream source node with the same Amazon node. We repeat each experiment with three source rates: 10 kbps (low), 100 kbps (medium) and 1 Mbps (high). The experiments are conducted over a 24-hour period, computing averages to avoid undesired effects due to variations in Internet traffic.

[1]Typically, one EC2 compute unit provides the equivalent CPU capacity of a 1.0–1.2 GHz 2007 AMD Opteron or a 2007 Intel Xeon CPU.

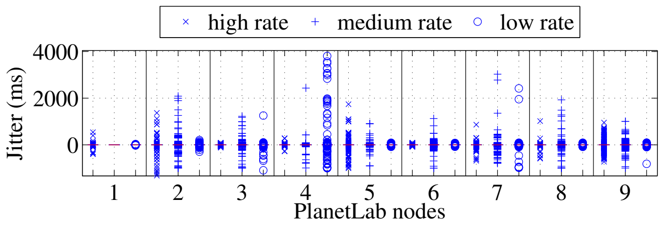


Fig. 2. Jitter experienced when sending streams to Amazon EC2

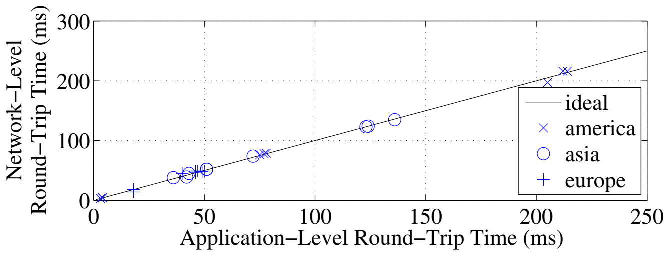


Fig. 3. Comparison of network- and application-level delay on Amazon EC2

Results. Fig. 2 shows the average jitter experienced by packets sent from each PlanetLab source to the engine. We observe that the average jitter is less than 2.5 μs, although some outliers exhibit a value of almost 4 seconds. To compensate for this, a DSPS would need to use a buffer size of at least this size, increasing overall system latency, or to discard these items. The observed level of jitter appears independent of the stream rate, and we believe that it is dominated by network effects.

In terms of application-level delay, we show the relationship between measured network-level and application-level round-trip delay in Fig. 3. Application-level round-trip times are obtained by comparing the timestamps when a tuple is sent by the source and when the result tuple is received by the same source after processing by the engine. Network-level round-trip times are reported based on ICMP ping messages between the two machines. We see that the cloud DC does not cause application-level delay to increase. It is instead dominated by the network latency of the wide-area Internet paths.

B. Processing measurements

Next we benchmark processing performance. We deploy the *Esper* stream processing engine [22] with several types of VMs available on Amazon EC2, and we explore if there is performance variation correlated with the time-of-day, affecting processing latency and throughput.

Esper processing engine. Esper processes continuous streams of time-based data. We use the native Esper benchmark tool that is composed of a data generator and a submitter. It generates a stream of shares and stock values for a given symbol at a fixed rate, set as a configuration parameter. In our case, we set the rate to be 30,000 stock values sent per second, generating a stream with 1000 different symbols. A query, which is defined in the Esper language, calculates the maximum stock value of a symbol in each second.

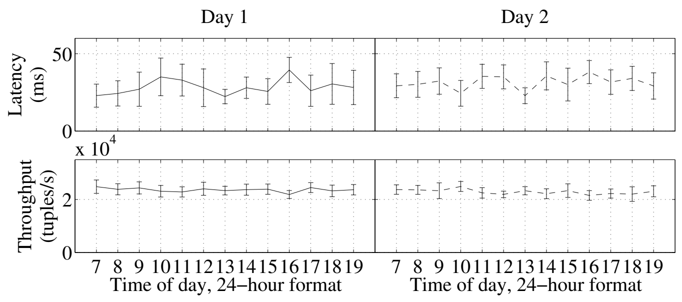


Fig. 4. Performance of Esper as a function of time on Amazon EC2

Since our goal is to explore if time-of-day variation affects processing latency when VMs are overloaded, we separate processing from network latency by placing all nodes in the same Amazon EC2 DC located in Virginia, US. We carry out 10 runs, executing processing engines in small VM instances with 1.7 GB of memory and 1 EC2 compute unit. The submitter machines are extra large instances with 15 GB of memory and 8 EC2 compute units. Each submitter sends 30,000 tuples/second to the engine to potentially overload it.

Results. The results in Fig. 4 show the variation of processing latency and throughput during different times over two week days. The observed throughput remains relatively stable over the measurement period, although latency suffers more from unpredictable outliers. We did not find an obvious pattern that correlated cloud performance with time-of-day for best-effort VMs, i.e. there is no significant variation consistent across multiple week days. We leave a study over longer time intervals for future work.

Next we examine how cloud VMs support increasing input data rates. We manually increase the number of VMs running Esper engines, while also gradually changing the stream rate from 100,000–200,000 tuples/second. The Esper submitter is deployed on an extra-large EC2 instance. When a VM is close to saturation, i.e. the engine cannot process all incoming data, we manually add another VM of the same type.

Fig. 5 on the left plots the throughput when using small VM instances and large instances are presented on the right. Different patterns represent different VMs of both instance types. We can see that the cloud VMs can be used to scale efficiently with an increasing input rate, achieving higher throughput. The number of VMs required to obtain a given throughput value depends on their type: as expected, fewer large than small instances are needed to achieve the same throughput. While running these scenarios we did not experience any significant degradation in the CPU performance and throughput that could be explained by VM co-location.

C. Discussion

We considered metrics related to cloud-deployed stream processing systems. First, end-to-end latency is composed of network and processing latency. Our results show that network latency is dominated by the geographic distance between the cloud DC and the sources, and the virtualised cloud infrastruc-

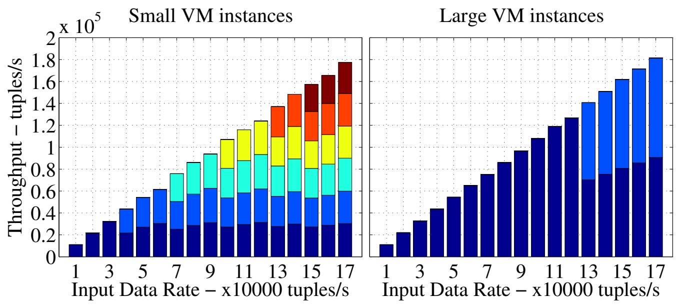


Fig. 5. Increase in throughput with different instance sizes on Amazon EC2 (Different shades/colours correspond to different VMs.)

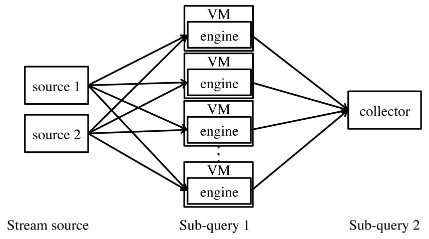


Fig. 6. Elastic DSPS with query partitioning across processing VMs

ture itself does not increase end-to-end latency significantly. Therefore, it is preferable to deploy stream processing engines at cloud sites within close network proximity to sources.

Second, it is important to consider that jitter suffers from high outliers that can be orders of magnitude above the average. Typically systems compensate for jitter through buffering or discarding of late-arriving data. In our experiments, discarding delayed data items would have resulted in a small percentage of lost data (approx. 3%).

In summary, when deploying a DSPS in a public cloud, it is necessary to understand the trade-offs when scaling to different numbers of VMs. A challenging issue is to decide on the right number of VMs and their instance types to support a given stream processing workload. After deployment, it is necessary to monitor the performance of processing VMs, and if they show decreasing throughput, to scale out to more VMs.

IV. Adaptive Cloud Stream Processing

We now present an adaptive algorithm to scale the number of VMs required to deploy a DSPS in the cloud. Our goal is to build an elastic stream processing system that resizes the number of VMs in response to input streams rates. The goal is to maintain low latency with a given throughput, while keeping VMs operating to their maximum processing capacity. We assume that a workload can be partitioned among multiple VMs, balancing streams equally across them. We also assume that there are always sufficiently many VMs available to scale up to workload demands.

As shown in Fig. 6, we assume that the DSPS executes a query, which can be decomposed across multiple VMs by splitting the query into sub-queries, each processing a sub-stream on a given engine. The input stream can be equally partitioned into sub-streams. For example, queries that compute aggregate and topK functions are naturally decomposable in this fashion. The results from sub-queries are then sent to a collector that merges them by executing another sub-query, emitting the overall query result. We further assume that load shedding is employed by the DSPS in overloaded conditions to sustain low-latency processing.

Algorithm 1 Adaptive provisioning of a cloud-based DSPS

```
Require: $totalInRate$, $N$, $maxRatePerVM$
Ensure: $N'$ s.t. $projRatePerVM * N' = totalInRate$
 1: $expRatePerVM = \lfloor totalInRate/N \rfloor$
 2: $totalExtraRateForVMs = 0$; $totalProcRate = 0$
 3: for all deployed VMs do
 4:     $totalExtraRateForVMs$ += $expRatePerVM$ - $getRate(VM)$
 5:     $totalProcRate$ += $getRate(VM)$
 6: end for
 7: $avgRatePerVM = \lfloor (totalProcRate/N) \rfloor$
 8: if $totalExtraRateForVMs > 0$ then
 9:     $N' = N + \lceil (totalExtraRateForVMs/avgRatePerVM) \rceil$
10:     $maxRatePerVM = avgRatePerVM$
11: else if $totalExtraRateForVMs < 0$ then
12:     $N' = \lceil totalInRate/maxRatePerVM \rceil$
13: end if
14: $projRatePerVM = totalInRate/N'$
15: return $N'$
```

Our proposed provisioning algorithm uses a black-box approach, i.e. it is independent of the specifics of queries running in the DSPS. It scales the number of VMs used solely based on measurements of input stream rates. It detects an overload condition when a decrease in the processing rate of input data occurs because of discarded data tuples due to load-shedding. The algorithm is invoked periodically and calculates the new number of VMs that are needed to support the current workload demand. This number can be larger than (when the system is overloaded and requires more resources), smaller than (when the system has spare capacity) or equal to the current number of engines. The aim is to maintain the required number of VMs, operating almost at their maximum capacity.

A. Algorithm

We present the provisioning algorithm more formally in Alg. 1. The algorithm takes as input the aggregate rate of the input stream, $totalInRate$, and the number of VMs currently used by the DSPS, N. It also takes $maxRatePerVM$, which is the maximum rate that a single VM can process, from previous invocations based on measurements in overload conditions. The algorithm takes a conservative approach, in which it gradually increases the number of VMs to reach the required set for sustainable processing. The output of the algorithm is the number of VMs, N', that is needed to sustain $totalInRate$. In this case, $totalInRate$ is divided equally among VMs and each handles $projRatePerVM$.

The algorithm initially estimates the stream rate each VM is expected to process, $expRatePerVM$, given the number of running VMs and the total input rate (line 1). $expRatePerVM$ is used to decide if the current partitioning of the input stream across VMs is adequate to support $totalInRate$. To this end, the algorithm calculates the difference between expected and processed data rates across VMs, $totalExtraRateForVMs$ (lines 3–5). Note that the function $getRate(VM)$ returns the current processing rate for a given VM. In line 6, the algorithm calculates the average processing capacity per VM, $avgRatePerVM$, in terms of processing input rate (line 6).

A positive value of $totalExtraRateForVMs$ (line 7) indicates that the current number of VMs N is inadequate to support the incoming $totalInRate$. Therefore, we need to increase the number of VMs to N', which is calculated by dividing the additional stream rate needed, $totalExtraRateForVMs$ by the average processing capacity per VM, $avgRatePerVM$ (line 8). At this point, the maximum processing capacity of each VM, $maxRatePerVM$, is updated (line 9). It is updated each time an overload condition is detected and maintained between invocations of the algorithm. If there is excess of processing capacity, i.e. $totalExtraRateForVMs$ is negative, we scale down VMs (lines 10-11). This is done by estimating the number of VMs required to support $totalInRate$, given that the maximum processing capacity per VM is $maxRatePerVM$ (line 11). Note that our approach assumes that the input stream is equally partitioned across VMs. The stream rate sent to each VM is given by $projRatePerVM$ (line 12).

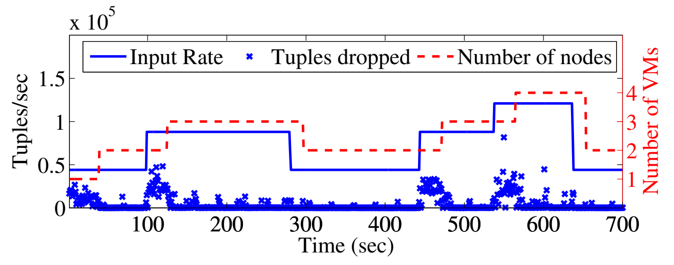
B. Implementation

We have implemented a prototype version of the adaptive algorithm as a shell script. It is integrated with the Esper processing system engine and uses a framework to control VMs and to collect required performance metrics. Performance metrics, i.e. throughput, processing latency and network latency, are generated by Esper engines and stored locally in a log file. The algorithm gathers them by remotely accessing the logs from each engine. Esper engines are started and stopped through standard calls to a management interface. We deployed this enhanced version of Esper on Amazon EC2. Based on our experience, this approach can be easily integrated with other cloud environments or DSPSs.

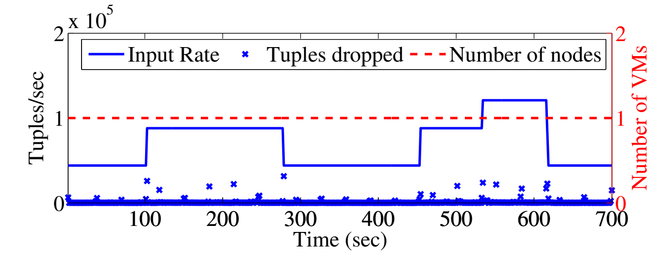
V. Experimental Evaluation

The goal of the evaluation is to illustrate the effectiveness of our adaptive provisioning algorithm for scaling the number of required VMs against variable input rates.

Experimental set-up. We use the same experimental setup as in §III-B with Esper and data streams obtained from its benchmark tool. We implement the Esper tuple submitter and vary the input tuple rate in a step-wise way as shown by the solid line in Fig. 7. Such variations in the input rate emulate demanding and sharp changes in the workload, similar to workloads adopted by others in dynamic resource provisioning [23]. We use two submitter VMs to send data, which are deployed on extra-large Amazon EC2 instances. We perform two sets of experiments with the Esper system



(a) Small EC2 instances



(b) Large EC2 instances

Fig. 7. Dynamic adaptation of VM numbers based on changes in input rates

deployed on small and large VM instances. In all cases, the VMs are created in advance to isolate the algorithm response time from VM set-up times, which can take several minutes.

The used query computes the maximum value of a stock with a given symbol per time window. To control the number of engines, we divide the query into two stages, as shown in Fig. 6. This allows us to use multiple engines to process the data in the first stage that takes the maximum value of each stock symbols per second. In the second stage, the first-stage results are collected and merged to obtain the final result.

A key parameter of our algorithm is the time interval between successive invocations. A system with a short interval would react quickly to varying input rates. However, it might obtain incorrect throughput estimates caused by transient behaviour. A large interval would mean that the system can only react slowly to changes in the input rate, staying in an over- or under-loaded condition.

Based on our workloads, we determined empirically the time interval as follows. The Esper engine calculates the number of dropped tuples every second. Using Fig. 7(a), which shows a large variation in the number of dropped tuples when the input rates vary and the system is overloaded, we determined that an invocation period of 30 seconds gives a good trade-off between these two extremes.

Results. As the input rate varies, Figs. 7(a) and 7(b) show the number of VMs allocated using small and large EC2 instances, respectively. The solid lines depict the input data rates and the dashed lines represent the number of VMs allocated by the adaptive algorithm. Dropped tuples due to overload are shown as stars. The processing latency maintained remains low (i.e. 7–28 μs) throughout the entire experiment because the Esper engine drops data tuples when overloaded.

When using small VM instances, the system scales up and down the number of VMs as required by the input rate. During a rate increase, we observe that the rate of dropped tuples increases until the algorithm detects the change and allocates more VMs. In the case of large instances, the algorithm sustains processing with a single VM because it is enough to handle the rate changes, with few tuples dropped throughout.

While our adaptive algorithm is able to scale the number of VMs against the input rate changes, there is room for improvement. There is a significant reaction delay before VMs are scaled up and down. To reduce this delay, other performance parameters, such as the percentage of idle CPU time or the variance in dropped tuples, could also be considered. Another challenge is that, in practice, it may take several minutes to allocate extra VMs. A workaround may be to reduce this time by exploiting Amazon EC2's facility for creating point-in-time snapshots of volumes using Elastic Block Storage. If used as a starting point for new VMs, it can reduce boot-up time.

VI. Future Work

We propose multiple future research directions to adapt existent DSPS engines to a cloud computing paradigm. We want to explore a wider range of VM types, monitoring performance over longer periods of time and switching between VM types. We also plan to test our approach in other cloud environments.

Second, we want to explore how to predict the future behaviour of input data rates and use this as part of our decision making. This would allow us to proactively change the number of processing VMs based on statistical workload variations. In addition, we want to explore if latency measurements can be used as an early predictor of overload conditions.

Third, we believe that there is scope for more advanced autonomic techniques for resource provisioning with minimum human intervention similar to [23], [24]. We plan to investigate if VM parameters in a cloud-based DSPS can be controlled dynamically based on continuous performance measurements, taking actions when there are violations in the performance.

VII. Conclusions

Stream processing systems are becoming largely important as the number of data-intensive applications increase. Cloud-based DSPSs offer unique opportunities for scalable data processing against time-changing stream rates. Based on our experimental evidence, public clouds are suitable deployment environments for stream processing. Our results show that latency is the dominating factor governing the performance of a cloud-based DSPS.

To answer the question of how to provision a DSPS in a public cloud given a workload, we propose an adaptive algorithm that scales a DSPS deployment in response to runtime variations of input stream rates. Our algorithm periodically estimates the number of VMs required to support the input stream data rate such that none of the VMs is overloaded. We evaluate this algorithm experimentally by deploying it as part of the Esper stream processing system on the Amazon EC2 cloud. Results show that our approach can scale up and down the number of DSPS engines on VMs as input rates change and maintain low processing latency with low data loss.

Acknowledgements

This work was supported by grant EP/F035217/1 ("DISSP: Dependable Internet-Scale Stream Processing") from the UK Engineering and Physical Sciences Research Council (EPSRC).

References

[1] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, "Gigascope: A Stream Database for Network Applications," in *SIGMOD*, 2003.

[2] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel *et al.*, "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *SOSP*, 2003.

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

[5] B. Babcock, M. Datar, and R. Motwani, "Load Shedding for Aggregation Queries over Data Streams," in *ICDE*, 2004.

[6] N. Tatbul, U. Çetintemel, S. Zdonik, M. Chemiack, and M. Stonebraker, "Load Shedding in a Data Stream Manager," in *VLDB*, 2003.

[7] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing," in *VLDB*, 2007.

[8] J. Wolf, N. Bansal, K. Hildrum, S. Parekh *et al.*, "SODA: An Optimizing Scheduler for Large-scale Stream-based Distributed Computer Systems," in *Middleware*, 2008.

[9] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh, "Robust Query Processing through Progressive Optimization," in *SIGMOD*, 2004.

[10] Y. Xing, S. B. Zdonik, and J.-H. Hwang, "Dynamic Load Distribution in the Borealis Stream Processor," in *ICDE*, 2005.

[11] Y. Zhou, B. C. Ooi, K.-L. Tan, and J. Wu, "Efficient Dynamic Operator Placement in a Locally Distributed Continuous Query System," in *OTM Conferences*, 2006.

[12] E. Kalyvianaki, W. Wiesemann, Q. H. Vu, D. Kuhn, and P. Pietzuch, "SQPR: Stream Query Planning with Reuse," in *ICDE*, 2011.

[13] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch, "Balancing Load in Stream Processing with the Cloud," in *SMDB*, 2011.

[14] "Amazon Elastic Compute Cloud (EC2)," accessed 2011. [Online]. Available: http://aws.amazon.com/ec2/

[15] J. Rehr, F. Vila, J. Gardner, L. Svec, and M. Prange, "Scientific Computing in the Cloud," *Computing in Science Engineering*, vol. 12, no. 3, 2010.

[16] G. Wang and T. S. E. Ng, "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center," *IEEE INFOCOM*, Mar. 2010.

[17] S. K. Barker and P. Shenoy, "Empirical Evaluation of Latency-sensitive Application Performance in the Cloud," in *ACM SIGMM conference on Multimedia systems*, ser. MMSys'10, 2010.

[18] K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright, "Performance Analysis of High Performance Computing Applications on the Amazon Web Services Cloud," in *CLOUDCOM*, 2010.

[19] C. Evangelinos, "Cloud Computing for Parallel Scientific HPC Applications: Feasibility of Running Coupled Atmosphere-Ocean Climate Models on Amazon's EC2." *CCA*, vol. 2, no. 2.40, 2008.

[20] S. Ostermann, R. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "An Early Performance Analysis of Cloud Computing Services for Scientific Computing," *TU Delft, Tech. Rep.*, Dec. 2008.

[21] J. Dittrich, J.-A. Quian, and J. Schad, "Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance," *VLDB Endowment*, vol. 3, 2010.

[22] T. Bernhardt, "Esper," Nov. 2011. [Online]. Available: http://esper.codehaus.org/

[23] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant, "Automated Control of Multiple Virtualized Resources," in *EuroSys*, 2009.

[24] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers using Kalman Filters," in *ICAC*, 2009.