Overload Management in Data Stream Processing Systems with Latency Guarantees

Evangelia Kalyvianaki*, Themistoklis Charalambous‡, Marco Fiscato*, and Peter Pietzuch*

*Department of Computing, Imperial College London, UK. E-mail: {ekalyv,mfiscato,prp}@doc.ic.ac.uk

‡Automatic Control Lab, Royal Institute of Technology (KTH), Stockholm, Sweden. E-mail: themisc@kth.se

ABSTRACT

Stream processing systems are becoming increasingly important to analyse real-time data generated by modern applications such as online social networks. Their main characteristic is to produce a continuous stream of fresh results as new data are being generated at real-time. Resource provisioning of stream processing systems is difficult due to time-varying workload data that induce unknown resource demands over time. Despite the development of scalable stream processing systems, which aim to provision for workload variations, there still exist cases where such systems face transient resource shortages. During overload, there is a lack of resources to process all incoming data in real-time; data accumulate in memory and their processing latency grows uncontrollably compromising the freshness of stream processing results. In this paper, we present a feedback control approach to design a nonlinear discrete-time controller that has no knowledge of the system to be controlled or the workload for the data and is still able to control the average tuple end-to-end latency in a single-node stream processing system. The results, of our evaluation on a prototype stream processing system, show that our method controls the average tuple end-to-end latency despite the time-varying workload demands and increasing number of queries.

1. INTRODUCTION

There is a high demand for real-time analysis of large volumes of streams of data generated by modern applications. For example, real-time data generated by the `Twitter` online social network, have shown to be applicable for financial predictions [24], rapid notification of earthquakes [5] and outcome prediction of UK national elections [16]. Additional application domains that benefit from real-time data analysis include, among others, healthcare and governmental administration, environmental sensing infrastructures and corporate business managed systems.

Over the last years, different *stream processing systems* (SPSs) have been developed to address the requirements of high-throughput and timely delivery of results in data streaming applications [26]. As new, *fresh* data are continuously generated by applications, it is important to process data in *near* real-time by keeping end-to-end latency low and thus providing up-to-date analysis. Furthermore, SPSs are required to cope with high, unknown and time-varying rates of fresh data in addition to increasing the number of users. These conditions pose unique challenges in the provisioning of resources to SPSs, e.g., CPU time.

In order to cope with demanding workloads *distributed* SPSs (DSPSs) are deployed over a set of nodes where each node hosts a single stream processing engine (SPE) and all collaborate transparently towards a resource powerful SPS; popular systems include Borealis [3], IBM's System S [12], Twitter Storm [1] and Yahoo S4 [2]. Recent approaches also address the problem of leveraging cloud resources to dynamically provision against time-varying workloads when required [17, 21, 8].

Nevertheless, conditions of resource saturation can still exist in a SPS; for example, during transient workload fluctuation or while allocating additional cloud resources for a workload with increasing demands [8]. When a SPS is overloaded, its resources, e.g., CPU time, are insufficient to process all incoming load. In this case, data are accumulated and await processing, causing the stream processing latency to grow. However, due to the freshness requirement of SPSs, increasing latency might render the stream processing results obsolete by the time they are returned to the user, even for short periods of overload.

In this paper, we address the problem of controlling the average stream processing latency in periods of resource overload. We assume that our approach benefits applications that can delay the *freshness* of data streaming result in order to fully utilise available resources. For example, consider an application that calculates the average number of tweets per user from North America per hour. In this case, delaying results by a few minutes does not have a significant impact on the utility of results. We assume that users provide us with a tolerance value of average latency for their applications.

Our approach to control latency is by discarding incoming data load via *load shedding*. Load shedding [4, 28, 23] is a well-studied technique in SPSs that involves the elimination of a portion of incoming data from processing to reduce the required resource footprint. Related approaches exist for single-node SPEs [4, 28] and DPSPs [35, 11, 22, 27]. We use random load-shedding [28] to randomly select data to discard among all awaiting processing in order to reduce the processing time of the remaining data and control the end-to-end latency.

It becomes apparent that when discarding incoming data, the values of the stream processing results change. Therefore, previous work in load shedding has proposed methods to minimise the deviation from perfect processing, i.e., without load shedding, using query approximation and semantic shedding techniques [10, 14, 18, 15, 23, 32, 13]. The above proposed strategies achieve their goals by leveraging extensive domain knowledge of the stream processing applications. Instead, we consider the streaming application as a black-box and by treating all data as equally important we randomly choose data to discard among all available. We assume that our system is used by applications, in which random shedding does not significantly change the values of the result outcome, e.g., average aggregate functions. In future work, we plan to apply semantic shedding for specific application domains in order to control latency while minimising performance loss.

The contributions of this work are the following: (a) we introduce a data shedding, nonlinear controller that chooses a proportion of data in the incoming queue such that the desired end-to-end latency remains within a target-interval, on average; our controller does not use any modelling knowledge about the system; (b) we discuss stability issues of the controller in the presence of inherent delays to the system; (c) we evaluate our controller using a prototype stream processing system and time-varying workloads and changing number of queries.

The rest of this paper is organised as follows. Section 2 presents the data streaming processing model considered in this paper. Section 3 presents the tuple shedding controller with which the average tuple processing latency is controlled. In Section 4 our approach is evaluated. Related work in the literature in discussed in Section 5. Finally, Section 6 summarises the contributions of this work and draws directions for future work.

2. DATA STREAM PROCESSING MODEL

In this section, we describe the data streaming model and our prototype SPS used to implement and evaluate our latency- and control-based load shedding approach.

Data Stream Model. We support a relational data stream model where data are organised into *tuples*. Each tuple contains a set of values of pre-defined types. A *stream* identifies an infinite set of time-ordered tuples. In data streaming, new tuples are continuously generated by *sources* and then processed according to the specifications of *user queries*. We support queries that are described by a tree of *operators*. The query tree represents the operators and their connections. We support operators with standard relational semantics, such as filter, join, min, max, count, average, top-k, group-by and time and count windows. The user also specifies the data sources and their connections to the query tree. An operator takes as input one or more streams and produces a single output stream. The stream emitted by the root operator of the query tree is the *query result*.

Stream Processing Engine. Figure 1 illustrates the Dependable Internet-Scale Stream Processing (DISSP) multi-component SPE which executes multiple queries at the same time. For every new query, a query coordinator component is created that connects the query to its sources and manages the query throughout its lifetime. As new tuples arrive from the sources, they are initially inserted in the incoming buffer (IB) queue. Tuples are then removed from the IB queue in a first-in-first-out order and passed for processing at the components executing the query operators. Once a tuple arrives at the DISSP node, its arrival time is attached to the tuple. As tuples are processed, their arrival times are propagated

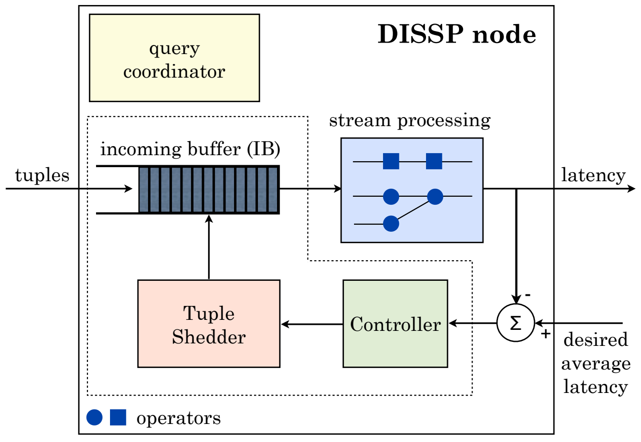


Figure 1: DISSP stream processing engine.

to the tuples of the result stream. When an operator takes more than one input tuples to produce an output tuple, the oldest arrival time from all the input tuples is forwarded to the output tuple. The *end-to-end latency* of tuples is the difference between the creation time of a result tuple at the query root operator and its arrival timestamp.

Problem Statement. It becomes apparent that if the arrival rate of tuples in the IB queue is higher than the rate at which tuples are passed for stream processing, the size of the IB continuously grows until eventually the SPE runs out of memory resources to allocate for new incoming tuples in IB. *In this paper, we address the problem of computing the number of tuples to randomly discard from the IB such that the average end-to-end latency over time across tuples and queries approaches a user-defined target.* Although the end-to-end tuple latency is the sum of the waiting and processing times, in case of overload it is dominated by the waiting time. Therefore, our system aims to control the waiting time by controlling the number of tuples to keep in the IB.

Feedback Control Loop. Figure 1 also illustrates the feedback control loop and the controller component within DISSP. Periodically, at every fixed interval, the IB queue is locked and the controller component using the control law decides the number of tuples to keep in the queue. This number is forwarded to the *tuple shedder component*, which randomly selects tuples to keep in the IB until the total number of tuples kept reaches the number calculated by the controller. The tuple shedder then discards all the remaining tuples from the IB. Before discarding tuples, the shedder performs bookkeeping tasks for each tuple in the IB; for example, the shedder updates the number of tuples discarded across queries to provide feedback to the users regarding the loss of data due to shedding. Therefore, there is an *information gathering cost* for each tuple. Once this process is completed, the IB queue is unlocked and the remaining tuples are forwarded to query operators; there is also a *processing cost* for each tuple. Note that, while the IB is locked, we keep any incoming new tuples in a secondary queue (not shown in the figure). The tuples in the secondary queue are appended at the end of the IB queue once the latter is unlocked.

3. TUPLE SHEDDING CONTROLLER

When designing a system, it is important to create a mathematical model of the plant in order to be able to provide accurate controllers and better performance. However, due to the complexity of stream processing applications and their time-varying dynamics, it is difficult to derive a precise mathematical model. More specifically, the information gathering and processing costs for tuple across queries may differ and the rates at which tuples arrive may change. Hence, it is very challenging to accurately model the costs of individual tuples.

Previous work in the field, e.g., [31], considered system identification techniques. While these techniques are powerful and contribute to the experimental study of the system's dynamics, they need offline training and cannot adapt to changing dynamics easily. In [31] the importance of feedback control is enunciated and a comparison of feedback versus non-feedback control strategies is discussed.

In the discrete time set-up that we investigate, we conveniently define the time coordinate so that unity is the time between consecutive iterations of the controller. We denote $c_s(t)$ and $c_p(t)$ to be the information gathering and processing costs for each tuple, respectively, at the time $t = 0, 1, 2, \ldots$. The problem being targeted is to create a controller that at each time instant chooses a number of tuples to be processed, denoted by $n(t)$, of the random number of tuples in the IB queue, denoted by $N(t)$, such that, the desired end-to-end latency remains within the time-interval T on average.

The latency of the system at time t, denoted by $\ell(t)$, is given by

$$\ell(t) \triangleq N(t)c_s(t) + n(t)c_p(t), \tag{1}$$

i.e., there is a cost for all tuples in the IB queue for information gathering and a cost of the tuples being processed.

In our approach, assuming that the target interval T is kept constant (i.e., $T(t) = T \ \forall t$), we require that the latency of the system remains smaller than or equal to the target interval T, i.e., $l(t) \leq T$ for all times t. Note that we can choose $n(t)$, such that $N(t)c_s(t) + n(t)c_p(t) \ll T$, but the system will operate well below its capacity, which is undesirable (see Figure 2). However, if we choose $n(t)$ such

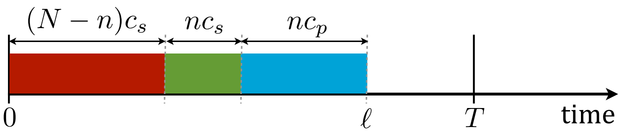


Figure 2: Latency ℓ remains less than the desired latency T, meaning that useful resources remain unexploited.

that $N(t)c_s(t) + n(t)c_p(t) \gg T$, the latency $\ell(t)$ exceeds the desired latency T due to the fact that there are many tuples in the queue (see Figure 3).

Let $n^\star$ be the optimal number of elements, such that our system will be able to process all $n^\star$ within the time-interval T, i.e., $n^\star = \{n(t) : l(t) = T\}$. Hence,

$$N(t)c_s(t) + n^\star(t)c_p(t) = T. \tag{2}$$

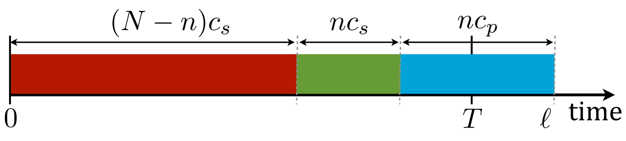


Figure 3: Latency ℓ exceeds the desired latency T; the constraint of having $\ell < T$ is soft and may be violated for some time instances.

Therefore, our target is to reach the optimal number of tuples to be processed provided that the costs are unknown and change over time, and we only measure the latency of the system with some delay that corresponds to the actual latency $\ell(t)$, and hence is of the order of the user-desired target latency T. The system dynamics are shown in Figure 4, where the tuples to be processed are controlled and the latency is the output of the system.

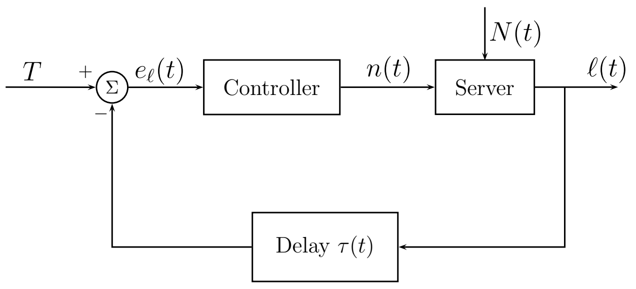


Figure 4: Block diagram of the closed-loop control system.

In this work, we first consider a simple integral controller of the following form

$$n(t+1) = n(t) + pe(t - \tau(t)), \tag{3}$$

where p is a positive constant, $e(t) \triangleq n^\star(t) - n(t)$ is the error from the optimal value at time t, and $\tau(t)$ is a time-varying bounded delay (i.e., $0 \leq \tau(t) \leq \tau_{\max}$ for all t). Using equation (2), the error $e(t)$ can be expressed as

$$\begin{aligned} e(t) &= \frac{T - N(t)c_s(t)}{c_s(t) + c_p(t)} - n(t) \\ &= \frac{T - N(t)c_s(t) - n(t)(c_s(t) + c_p(t))}{c_s(t) + c_p(t)} \\ &= \frac{T - \ell(t)}{c_s(t) + c_p(t)}. \end{aligned}$$

If the costs $c_s(t)$ and $c_p(t)$ were constant and known throughout the operation of the stream processing system, our controller would be a simple linear integrator. However, these costs are unknown and more importantly they change according to the tuples inserted in the IB queue. In addition, since the measured latency is delayed, the controller will be given by

$$n(t+1) = n(t) + p\frac{T - \ell(t - \tau(t))}{c_s(t - \tau(t)) + c_p(t - \tau(t))}. \tag{4}$$

Since we do not have a measure of $c_s(t-\tau(t)) + c_p(t-\tau(t))$, we need to approximate it, but still make sure that the system remains stable. Hence, the update depending on the error should be equal or smaller to the actual error. In order to do this, we make use of the following remark.

REMARK 1. *Since $n(t) < N(t)$ for all times t, by* (1) *it is easily shown that*

$$\frac{n(t)}{\ell(t)} \leq \frac{1}{c_s(t) + c_p(t)} \leq \frac{N(t)}{\ell(t)} \qquad (5)$$

Therefore, our controller becomes of the form

$$n(t+1) = n(t) + pn(t-\tau(t))\frac{T - \ell(t-\tau(t))}{\ell(t-\tau(t))}, \qquad (6)$$

and will remain stable by an appropriate choice of constant p. For simplicity and for obtaining a less oscillatory controller, we approximate

$$\frac{n(t-\tau(t))}{\ell(t-\tau(t))} \approx \frac{n(t)}{T},$$

while accounting for the case where $n(t-\tau(t)) < n(t)$ and $\ell(t-\tau(t)) > T$, by choosing a control gain q, such that

$$q\frac{n(t)}{T} \leq p\frac{n(t-\tau(t))}{\ell(t-\tau(t))}.$$

Hence, controller (6) now becomes

$$n(t+1) = n(t) + qn(t)\frac{T - \ell(t-\tau(t))}{T}. \qquad (7)$$

Therefore, the problem boils down to choosing q such that the system remains stable. In linear discrete-time systems, the upper bound of the control gain is a decreasing function of the delay (see for example [20]).

PROPOSITION 1. *A system of the form*

$$x(t+1) = x(t) + ke(t-\tau(t))$$

for $0 \leq \tau(t) \leq \tau_{\max}$ *for all* t, *is stable provided that the control gain* k *is selected such that*

$$0 < k < \frac{2}{2\tau_{\max} + 1}.$$

Proposition 1 suggests how the control gain should be adapted in the case of delays. The controller proposed in (7) is non-linear, but it can be easily deduced that by choosing a small enough q the system is stable, provided the corresponding linear system is stable (using Proposition 1).

4. EVALUATION

In this section, we evaluate our approach to control the average tuple processing latency. We use the DISSP prototype SPE deployed on a single server machine. To create overload conditions we simulate a demanding workload of increasing number of user queries. Each user issues the same `avg` query that calculates *the average CPU consumption every second over ten server machines from the PlanetLab network.* We use ten source processes that generate data from real-world traces of resource utilisations of PlanetLab nodes [30]. Each source process emits tuples node at a time-varying rate by changing the submission rate every ten seconds in a repeated pattern: the rate increases from 50 to 150 and then decreases again to 50 in steps of 50. Such a workload pattern tests the controller against time-changing workloads. We

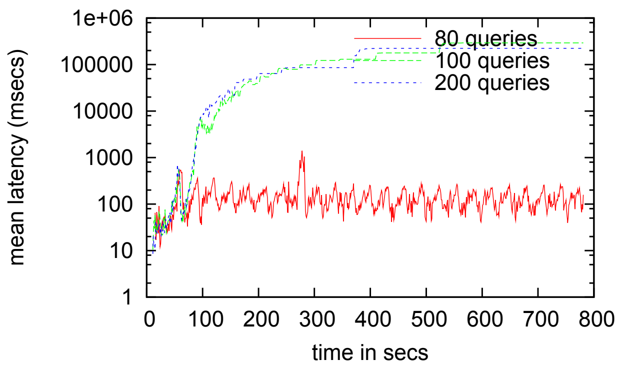


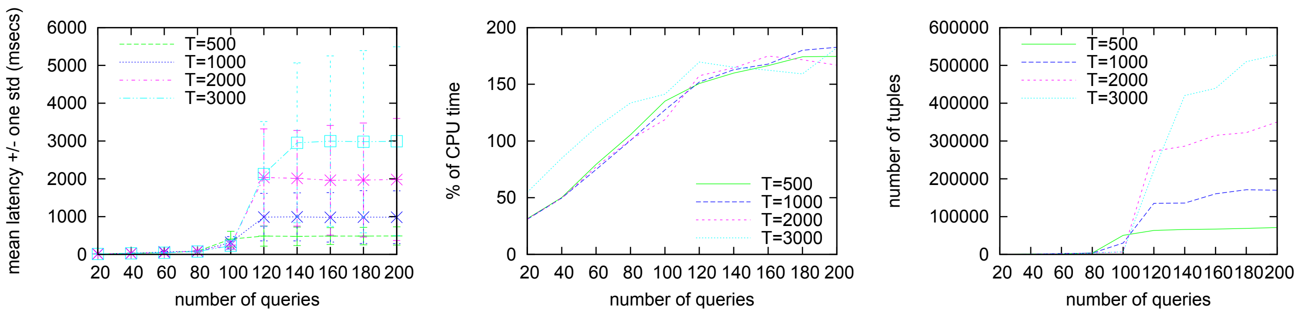
Figure 5: SPE performance without the controller.

use one server machine for the ten source processes and one machine for submitting the queries. All machines including the DISSP node are equipped with 4 CPU cores of 1.8 GHz each and 4 GB of memory, running Ubuntu Linux 2.6.27-17-server and are connected over 1 Gbps network.

Initially, we illustrate the rapid increase in the mean tuple latency when the controller and the tuple shedder are disabled and the SPE is overloaded. We perform different experiments, and for each one, we keep the number of queries constant. Overall we vary the number of queries from 20 to 200 in steps of 20. Figure 5 shows the mean tuple latency over time for each experiment for 80 (similar results hold for 20, 40 and 60 queries) and 100 and 200 queries (similar results hold for 120, 140, 160 and 180). The results show that, up to 80 queries, the SPE engine can cope with the incoming rate of tuples and processes all data for all queries adequately while keeping the latency low. However, once the number of queries exceeds 100, the latency increases rapidly until the engine runs out of memory to allocate for incoming tuples and the system can no longer execute after the first 450 seconds and beyond. To sustain more than 100 queries, the system needs to perform tuple shedding.

Figure 6 shows the mean tuple latency of the SPE when its incoming tuple load is managed by our latency-based controller. As before, we vary the number of queries and set four different target values for the mean tuple latency: T= 500, 1000, 2000 and 3000 *msecs*. In all cases, the controller is invoked every 250 *msecs* and the controller gain q is set to a small value of 0.1 to compensate for the various delays introduced by the target values. Results show that once the system overloads—when the CPU utilisation approaches 200% in Figure 6(b) above 120 queries—our controller manages to keep the mean latency at the target values in all four cases as shown in Figure 6(a). To reach and maintain the target value, the controller decides the number of tuples to keep in the IB, as shown in Figure 6(c). This figure shows that the number of tuples kept increases with the target value—with larger latency values the controller has to keep more tuples. This effect continues as the number of queries increases.

The goal of the controller is to maintain the mean tuple latency at a user-given target value. Although, as Figure 6(a) suggests, our controller performs well, there is a latency variation around the mean values. Figure 7 presents the empirical cumulative distribution function (CDF) for the different target values in the case of 200 queries (similar results hold when queries lie above 120). Although the mean value is in



(a) mean latency (b) CPU resources (c) tuples left

Figure 6: SPE performance with the shedding controller for different target values of mean query latency.

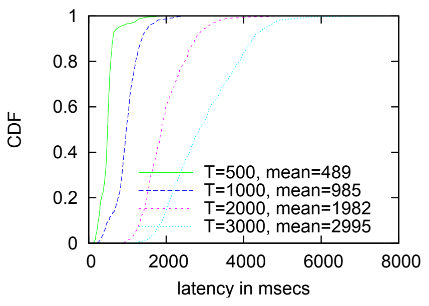


Figure 7: CDF of tuple latencies for 200 queries.

all cases very close to the target value, the variation around the mean increases with the target value. When the target value increases, the tuples remain in the IB for longer. As a result, a delay exists in the control loop which increases with the target value and affects the latency variance. To compensate for such delays we set q to the small value of 0.1.

5. RELATED WORK

Tu et al. [31] use system identification techniques to approximate the model of SPS, while initially assuming constant costs. Later, they allow for the costs to change, but very slowly compared to the data arrival rates, so that their identification techniques can capture the change. In addition, they ignore the delay that is inherent in such systems.

Load shedding has long been used to address resource overload in single-node SPS [4, 28, 23]. Tailored load shedding techniques exist for certain types of operators, such as joins [14, 13, 18, 10] and aggregations [6, 29]. Srivastava et al. [25] study the load shedding problem to reduce the memory footprint for windowed stream joins. Different techniques exist that limit the loss of information in the output stream caused by discarded data. For example, semantic shedding assumes a function that correlates tuples with their contribution to the query output [7, 28]. Tuples are discarded in ways to maximise query performance. Additionally, Wei et al. [32] apply semantic load shedding in XML streams.

In contrast, we use random load shedding [28] that is applicable to a wide-range of applications. We regard semantic shedding complementary to our approach. Rather than random shedding, our tuple shedder component can adopt another shedding technique. As our controller uses a simple, black-box approach to control latency, it can incorporate different shedding costs.

Query approximation techniques reduce the resource utilisations of operators by modifying the semantics of queries through query rewriting [23, 18, 10, 9, 33]. We differ from these approaches in that we target overload management across a wide range of queries and our controller has no control over the query semantics.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we present a feedback control approach to control the average end-to-end tuple latency in single-node SPSs. We have developed a non-linear controller to control the latency without accurately modelling the processing and information gathering costs. Our results show that our controller maintains the average latency very close to the target values across different number of queries and time-varying input tuple rates.

Controlling the end-to-end latency in SPSs is critical for stream processing users to maintain their latency requirements for their applications. In future work, we plan to include heterogeneous workloads with different types of queries. As resource demands vary across queries, it is important for a multi-tenant SPSs to control the resource allocations across queries while maintaining latency guarantees within queries of the same application class and also to address saturation of other resources such as memory.

We have designed and tested our control system for single-node SPSs. We plan to extend our work for DSPSs to control latency across queries and nodes. In multi-tenant DSPSs, where queries span many nodes and nodes are shared by queries, it is particularly challenging to control end-to-end tuple processing latency. We plan to investigate distributed strategies to build protocols where local, per-node decisions can make all queries of the DSPS to converge to user-defined latencies. This can be achieved by leveraging on the co-location of queries over a single node and the distribution of queries across nodes.

Load shedding in SPSs share the same goals with admission control in web servers environments, e.g., [19, 34], since both approaches aim to control the amount of processing load to tackle overload. In future work we plan to compare our approach with existing approaches from the area of admission control.

Acknowledgements. This work was supported by the grants: EP/F035217/1 (“DISSP: Dependable Internet-Scale

Stream Processing"), EP/I012036/1 ("Custom Computing for Advanced Digital Systems") and EP/E025188/1 ("AEDUS2: Adaptable Environments for Distributed Ubiquitous Systems") from the UK Engineering and Physical Sciences Research Council (EPSRC).

7. REFERENCES

[1] Twitter storm. https://github.com/nathanmarz/storm/wiki.

[2] Yahoo s4. http://incubator.apache.org/s4/.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[4] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a New Model and Architecture for Data Stream Management. *The VLDB Journal*, 12(2):120–139, 2003.

[5] Agence French Presse (AFP). Twitters beat media in reporting China earthquake, 2008.

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

[7] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: a New Class of Data Management Applications. In *VLDB*, pages 215–226. VLDB Endowment, 2002.

[8] J. Cervio, K. Evangelia, J. Salvacha, and P. Pietzuch. Adaptive Provisioning of Stream Processing Systems in the Cloud. In *SMDB*, 2012.

[9] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim. Approximate Query Processing using Wavelets. *The VLDB Journal*, 10:199–223, 2001.

[10] A. Das, J. Gehrke, and M. Riedewald. Approximate Join Processing over Data Streams. In *SIGMOD*, pages 40–51, New York, NY, USA, 2003. ACM.

[11] H. Feng, Z. Liu, C. H. Xia, and L. Zhang. Load Shedding and Distributed Resource Control of Stream Processing Networks. *Perform. Eval.*, 64(9-12):1102–1120, 2007.

[12] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S Declarative Stream Processing Engine. In *SIGMOD'08*, pages 1123–1134. ACM, 2008.

[13] B. Gedik, K.-L. Wu, P. Yu, and L. Liu. GrubJoin: An Adaptive, Multi-Way, Windowed Stream Join with Time Correlation-Aware CPU Load Shedding. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1363 –1380, 2007.

[14] B. Gedik, K.-L. Wu, P. S. Yu, and L. Liu. Adaptive Load Shedding for Windowed Stream Joins. In *CIKM*, pages 171–178, 2005.

[15] L. Golab and M. T. Özsu. Issues in Data Stream Management. *SIGMOD Rec.*, 32:5–14, June 2003.

[16] guardian.co.uk. Twitter election predictions are more accurate than YouGov.

[17] V. Gulisano, R. Jimenez-peris, M. Patino-martinez, and P. Valduriez. StreamCloud: A Large Scale Data Streaming System. In *ICDCS*, pages 126–137, 2010.

[18] J. K. Jeffrey, J. F. Naughton, and S. D. Viglas. Evaluating Window Joins over Unbounded Streams. In *ICDE*, pages 341–352, 2003.

[19] A. Kamra. Yaksha: a Self-tuning Controller for Managing the Performance of 3-tiered Web Sites. In *IWQoS*, pages 47–56, 2004.

[20] C.-Y. Kao and B. Lincoln. Simple Stability Criteria for Systems with Time-varying Delays. *Automatica*, 40(8):1429 – 1434, 2004.

[21] W. Kleiminger, E. Kalyvianaki, and P. Pietzuch. Balancing Load in Stream Processing with the Cloud. In *SMDB*, pages 16–21, 2011.

[22] Z. Liu, A. Tang, C. Xia, and L. Zhang. A Decentralized Control Mechanism for Stream Processing Networks. *Annals of Operations Research*, 170:161–182, 2009.

[23] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. S. Manku, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Resource Management, and Approximation in a Data Stream Management System. In *CIDR*, 2003.

[24] E. J. Ruiz, V. Hristidis, C. Castillo, A. Gionis, and A. Jaimes. Correlating Financial Time Series with Micro-blogging Activity. In *WSDM*, pages 513–522, 2012.

[25] U. Srivastava and J. Widom. Memory-limited Execution of Windowed Stream Joins. In *VLDB*, pages 324–335. VLDB Endowment, 2004.

[26] M. Stonebraker, U. Cetintemel, and S. Zdonik. The 8 Requirements of Real-time Stream Processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[27] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *VLDB*, pages 159–170. VLDB Endowment, 2007.

[28] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB*, pages 309–320. VLDB Endowment, 2003.

[29] N. Tatbul and S. Zdonik. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *VLDB*, pages 799–810. VLDB Endowment, 2006.

[30] The PlanetLab Consortium. PlanetLab. `http://www.planetlab.org`, 2004.

[31] Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load Shedding in Stream Databases: a Control-based Approach. In *VLDB*, pages 787–798. VLDB Endowment, 2006.

[32] M. Wei, E. A. Rundensteiner, and M. Mani. Utility-driven Load Shedding for XML Stream Processing. In *WWW*, pages 855–864, 2008.

[33] M. Wei, E. A. Rundensteiner, and M. Mani. Achieving High Output Quality Under Limited Resources Through Structure-based Spilling in XML Streams. *Proceedings VLDB Endow.*, 3:1267–1278, 2010.

[34] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *USITS*, pages 4–4, 2003.

[35] H. C. Zhao, C. H. Xia, Z. Liu, and D. Towsley. A Unified Modeling Framework for Distributed Resource Allocation of General Fork and Join Processing Networks. In *SIGMETRICS*, pages 299–310, 2010.