Délestage avisé dans les systèmes de traitement de flux

Nicolò Rivetti, Yann Busnel, Leonardo Querzoni

Délestage avisé dans les systèmes de traitement de flux

Nicoló Rivetti[1], Yann Busnel[2,3] et Leonardo Querzoni[4]

[1]*Technion - Israel Institute of Technology, Haifa, Israel*
[2]*IMT Atlantique, Rennes, France*
[3]*Inria Rennes – Bretagne Atlantique, France*
[4]*Sapienza University of Rome, Italy*

Le délestage de charge est une technique utilisée par les systèmes de traitement de flux en réaction aux pics de charge imprévisibles en entrée, lorsque les ressources de calcul ne sont pas suffisamment provisionnées. Le rôle du délesteur est d'abandonner certains tuples pour maintenir la charge en entrée en dessous d'un seuil critique, et éviter le débordement des mémoires tampons menant *in fine* à la défaillance complète du système. Dans cet article, nous proposons Load-Aware Shedding (LAS), une solution de délestage de charge qui ne repose ni sur un modèle de coût prédéfini ni sur des hypothèses sur les temps d'exécution des tuples. LAS construit et maintient dynamiquement et efficacement un modèle de coût pour estimer, par l'utilisation d'agrégats, la durée d'exécution de chaque tuple avec des taux d'erreur d'approximation faibles et bornés. Cette estimation est utilisée par un délesteur proactif, localisé en amont de chaque opérateur, permettant de réduire la latence liée aux files d'attente par le délestage d'un nombre minimal de tuples. Nous avons prouvé que LAS est une (ε, δ)-approximation d'un délesteur temps-réel optimal. De plus, nous avons évalué son impact sur des applications de traitement de flux, en terme de robustesse et de fiabilité, par une large expérimentation sur la plateforme Microsoft Azure.

Mots-clefs : Traitement de flux; Délestage de charge; Algorithme d'approximation probabiliste; Evaluation de performances

Travaux co-financés par le projet ANR SocioPlug (ANR-13-INFR-0003) et le projet DeSceNt du Labex CominLabs (ANR-10-LABX-07-01).

1 Introduction

Distributed stream processing systems (DSPS) are today considered as a mainstream technology to build architectures for the real-time analysis of big data. An application running in a DSPS is typically modeled as a directed acyclic graph where data operators (nodes) are interconnected by streams of tuples containing data to be analyzed (edges). The success of such systems can be traced back to their ability to run complex applications at scale on clusters of commodity hardware. Correctly provisioning computing resources for DSPS however is far from being a trivial task. System designers need to take into account several factors: the computational complexity of the operators, the overhead induced by the framework, and the characteristics of the input streams. This latter aspect is often critical, as input data streams may unpredictably change over time both in rate and content. Bursty input load represents a problem for DSPS as it may create unpredictable bottlenecks within the system that lead to an increase in queuing latencies, pushing the system in a state where it cannot deliver the expected quality of service (typically expressed in terms of tuple completion latency). *Load shedding* is generally considered a practical approach to handle bursty traffic. It consists in dropping a subset of incoming tuples as soon as a bottleneck is detected in the system.

Existing load shedding solutions[1, 4] either randomly drop tuples when bottlenecks are detected or apply a pre-defined model of the application and its input that allows them to deterministically take the best shedding decision. In any case, all the existing solutions assume that incoming tuples all impose the same computational load on the DSPS. However, such assumption does not hold for many practical use cases.

The tuple execution duration, in fact, may depend on the tuple content itself. This is often the case whenever the receiving operator implements a logic with branches where only a subset of the incoming tuples travels through each single branch. If the computation associated with each branch generates different loads, then the execution duration will change from tuple to tuple. A tuple with a large execution duration may delay the execution of subsequent tuples in the same stream, thus increasing queuing latencies and possibly cause the emergence of a bottleneck.

On the basis of this simple observation, we introduce Load-Aware Shedding (LAS), a novel solution for load shedding in DSPS. LAS gets rid of the aforementioned assumptions and provides efficient shedding aimed at matching given queuing latency goals, while dropping as few tuples as possible. To reach this goal LAS leverages a smart combination of *sketch* data structures to efficiently collect at runtime information on the time needed to compute tuples and thus build and maintain a cost model that is then exploited to take decisions on when load must be shed. LAS has been designed as a flexible solution that can be applied on a per-operator basis, thus allowing developers to target specific critical stream paths in their applications.

In summary, the contributions provided by this paper are (i) the introduction of LAS, the first solution for load shedding in DSPS that proactively drops tuples to avoid bottlenecks without requiring a predefined cost model and without any assumption on the distribution of tuples, (ii) a theoretical analysis of LAS that points out how it is an (ε, δ)-approximation of the optimal online shedding algorithm and, finally, (iii) an experimental evaluation that illustrates how LAS can provide predictable queuing latencies that approximate a given threshold while dropping a small fraction of the incoming tuples.

2 System Model and Problem Definition

We consider a distributed stream processing system (DSPS) deployed on a cluster where several computing nodes exchange data through messages sent over a network. The DSPS executes a stream processing application represented by a *topology*: a directed acyclic graph interconnecting operators, represented by vertices, with data streams (DS), represented by edges. Data injected by source operators is encapsulated in units called tuples and each data stream is an unbounded sequence of tuples. Without loss of generality, here we assume that each tuple t is a finite set of key/value pairs that can be customized to represent complex data structures. To simplify the discussion, in the rest of this work we deal with streams of unary tuples each representing a single non negative integer value. We also restrict our model to a topology with an operator *LS* (*load shedder*) that decides which tuples of its outbound DS σ consumed by operator O shall be dropped. Tuples in σ are drawn from a large universe $[n] = \{1, \ldots, n\}$ and are ordered, *i.e.,* $\sigma = \langle t_1, \ldots, t_m \rangle$. Therefore $[m] = 1, \ldots, m$ is the index sequence associated with the m tuples contained in the stream σ. Both m and n are unknown. We denote with f_t the unknown frequency of tuple t, *i.e.,* the number of occurrences of t in σ. We assume that the execution duration of tuple t on operator O, denoted as $w(t)$, depends on the content of t. In particular, without loss of generality, we consider a case where w depends on a single, fixed and known attribute value of t. The probability distribution of such attribute values, as well as w, are unknown, may differ from operator to operator and may change over time. However, we assume that subsequent changes are interleaved by a large enough time frame such that an algorithm may have a reasonable amount of time to adapt. On the other hand, the input throughput of the stream may vary, even with a large magnitude, at any time. Let $q(i)$ be the queuing latency of the i-th tuple of the stream, *i.e.,* the time spent by the i-th tuple in the inbound buffer of operator O before being processed. Let us denote as $\mathcal{D} \subseteq [m]$, the set of dropped tuples in a stream of length m, *i.e.,* dropped tuples are thus represented in $\mathcal{D}$ by their indices in the stream $[m]$. Moreover, let $d \leq m$ be the number of dropped tuples in a stream of length m, *i.e.,* $d = |\mathcal{D}|$. We can define the average queuing latency as: $\overline{Q}(j) = \sum_{i \in [j] \setminus \mathcal{D}} q(i)/(j-d)$ for all $j \in [m]$. The goal of the load shedder is to maintain the average queuing latency smaller than a given threshold τ by dropping as less tuples as possible while the stream unfolds. The quality of the shedder can be evaluated both by comparing the resulting $\overline{Q}$ against τ and by measuring the number of dropped tuples d. More formally, we define the load shedding problem as follows: given a data stream $\sigma = \langle t_1, \ldots, t_m \rangle$, find the smallest set $\mathcal{D}$ such that $\forall j \in [m] \setminus \mathcal{D}, \overline{Q}(j) \leq \tau$.

3 Load Aware Shedding

Load-Aware Shedding (LAS) is based on a simple, yet effective, idea: if we assume to know the execution duration $w(t)$ of each tuple t in the operator, then we can foresee queuing times and drop all tuples that will cause the queuing latency threshold τ to be violated. However, the value of $w(t)$ is generally unknown. LAS builds and maintain at run-time a cost model for tuple execution durations. It takes shedding decision based on the estimation $\widehat{C}$ of the total execution duration of the operator: $C = \sum_{i \in [m] \setminus \mathcal{D}} w(t_i)$. In order to do so, LAS computes an estimation $\hat{w}(t)$ of the execution duration $w(t)$ of each tuple t. Then, it computes the sum of the estimated execution durations of the tuples assigned to the operator, *i.e.,* $\widehat{C} = \sum_{i \in [m] \setminus \mathcal{D}} \hat{w}(t)$. At the arrival of the i-th tuple, subtracting from $\widehat{C}$ the (physical) time elapsed from the emission of the first tuple provides LAS with an estimation $\hat{q}(i)$ of the queuing latency $q(i)$ for the current tuple. To enable this approach, LAS builds a sketch on the operator (*i.e.,* a memory efficient data structure) that will track the execution duration of the tuples it processes. When a change in the stream or operator characteristics affects the tuples execution durations $w(t)$, *i.e.,* the sketch content changes, the operator will forward an updated version to the load shedder, which will than be able to (again) correctly estimate the tuples execution durations. This solution does not require any *a priori* knowledge on the stream or system, and is designed to continuously adapt to changes in the input stream or on the operator characteristics.

3.1 LAS design

The operator maintains two `Count Min` [2] sketch matrices (Figure 1.A): the first one, denoted as $\mathcal{F}$, tracks the tuple frequencies f_t; the second one, denoted as $\mathcal{W}$, tracks the tuples cumulated execution durations $W_t = w(t) \times f_t$. Both `Count Min` matrices share the same sizes and **2**-universal hash functions. The latter is a generalized version of the `Count Min` providing (ε, δ)-additive-approximation of point queries on stream of updates whose value is the tuple execution duration when processed by the instance. The operator will update both matrices after each tuple execution. The operator is modeled as a finite state machine (Figure 2) with two states: START and STABILIZING. The START state lasts as long as the operator has executed N tuples, where N is a user defined window size parameter. The transition to the STABILIZING state (Figure 2.A) triggers the creation of a new snapshot $\mathcal{S}$. A snapshot is a matrix of size $r \times c$ where $\forall i \in [r], j \in [c] : \mathcal{S}[i,j] = \mathcal{W}[i,j] / \mathcal{F}[i,j]$. We say that the $\mathcal{F}$ and $\mathcal{W}$ matrices are stable when the relative error η between the previous snapshot and the current one is smaller than a configurable parameter μ. Then, each time the operator has executed N tuples, it checks whether $\eta \leq \mu$. **(i)** In the negative case $\mathcal{S}$ is updated (Figure 2.B). **(ii)** In the positive case the operator sends the $\mathcal{F}$ and $\mathcal{W}$ matrices to the load shedder (Figure 1.B), resets their content and moves back to the START state (Figure 2.C).

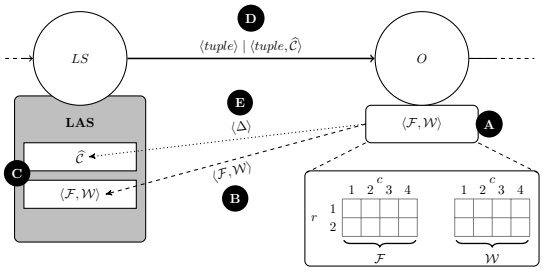

Fig. 1: Load-Aware Shedding data structures with $r = 2$ ($\delta = 0.25$), $c = 4$ ($\varepsilon = 0.70$).

The *LS* (Figure 1.C) maintains the estimated cumulated execution duration of the operator $\widehat{C}$ and a pairs of initially empty matrices $\langle \mathcal{F}, \mathcal{W} \rangle$. The load shedder computes, for each tuple t, the estimated queuing latency $\hat{q}(i)$ as the difference between the operator estimated execution duration $\widehat{C}$ and the time elapsed from the emission of the first tuple. It then checks if the estimated queuing latency for t satisfies the CHECK method. This method encapsulates the logic for checking if a desired condition on queuing latencies is violated or not. In this paper, as stated in Section 2, we aim at maintaining the average queuing latency below a threshold τ. Then, CHECK tries to add $\hat{q}$ to the current average queuing latency. If the result is larger than τ (**i**), it simply returns *true*; otherwise (**ii**), it updates its local value for the average queuing latency and returns *false*. Note that different goals, based on the queuing latency, can be defined and encapsulated within CHECK. If CHECK($\hat{q}$) returns *true*,

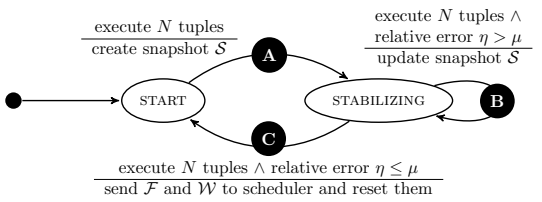

Fig. 2: Operator finite state machine.

(**i**) the load shedder returns *true* as well, *i.e.,* tuple t must be dropped. Otherwise (**ii**), the operator estimated execution duration $\widehat{C}$ is updated with the estimated tuple execution duration $\hat{w}(t)$, *i.e.,* the tuple must not be dropped. There is a delay between any change in $w(t)$ and when *LS* receives the updated $\mathcal{F}$ and $\mathcal{W}$ matrices. This introduces a skew in the cumulated execution duration estimated by *LS*. In order to compensate this skew, we introduce a synchronization mechanism (Figure 1.D and 1.E) that kicks in whenever the *LS* receives a new pair of matrices from the operator. Notice also that there is an initial transient phase in which the *LS* has not yet received any information from the operator. This means that in this first phase it has no information on the tuples execution times and is forced to apply some naive policy. This mechanism is thus also needed to initialize the estimated cumulated execution times in this initial transient phase.

The complete theoretical analysis of LAS is available in [3].

4 Experimental Evaluation

We extensively tested LAS both in a simulated environment with synthetic datasets and on a prototype implementation running with real data. Due to space constraints, the extensive simulation results are available in the companion paper [3]. To evaluate the impact of LAS on real applications we implemented a prototype targeting the Apache Storm framework. We have deployed our cluster on Microsoft Azure cloud service. We used a dataset containing a stream of preprocessed tweets related to the 2014 European elections. The tweets are enriched with a field *mention* containing the *entities* mentioned in the tweet. These entities can be easily classified into *politicians*, *media* and *others*. We consider the first 500,000 tweets, mentioning roughly $n = 35,000$ distinct entities. The test topology is made of a source (*spout*) and two operators (*bolts*) *LS* and *O*. The source reads the input stream and emits the tuples consumed by bolt *LS*. Bolt *LS* uses either Straw-Man, LAS or Full Knowledge to perform the load shedding on its outbound data stream consumed by bolt *O*. The Straw-Man algorithm uses the same shedding strategy of LAS, however it uses the average execution duration as the tuple's estimated execution duration. On the other hand, Full Knowledge knows the exact execution duration for each tuple. Finally operator *O* gather some statistics on each tweet and decorate the outgoing tuples with some additional information. However the statistics and additional informations differ depending on the class the entities mentioned in each tweet belong. Each of the 500,000 tweets may contain more than one mention, leading to 110 different execution duration values from 1 millisecond to 152 milliseconds. Figure 3 reports the average completion latency as the stream unfolds. As the plots show, LAS provides completion latencies that are extremely close to Full Knowledge, dropping a similar amount of tuples. Conversely, Straw-Man completion latencies are at least one order of magnitude larger. This is a consequence of the fact that in the given setting Straw-Man does not drop tuples, while Full Knowledge and LAS drop on average a steady amount of tuples ranging from 5% to 10% of the stream. These results confirm the effectiveness of LAS in keeping a close control on queuing latencies (and thus provide more predictable performance) at the cost of dropping a fraction of the input load.

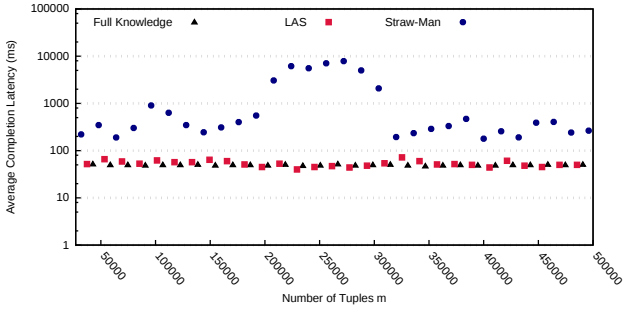

Fig. 3: Prototype use case: Average completion latency

References

[1] 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. *Int. J. on Very Large Data Bases*, 12(2), 2003.

[2] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. of Algorithms*, 55, 2005.

[3] N. Rivetti, Y. Busnel, and L. Querzoni. Load-aware shedding in stream processing systems. In *Proc. of the 10th ACM Int. Conf. on Distributed and Event-based Systems*, 2016.

[4] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 29th Int. Conf. on Very large data bases*, 2003.