AccStream: Accuracy-aware Overload Management for Stream Processing Systems

Haiyang Sun*, Robert Birke†, Walter Binder*, Mathias Björkqvist† and Lydia Y. Chen†
*Università della Svizzera italiana, Lugano, Switzerland. {haiyang.sun,walter.binder}@usi.ch
†IBM Research Zurich, Rüschlikon, Switzerland. Email: {bir,mbj,yic}@zurich.ibm.com

***Abstract*—With the rapid growth of social media and Internet-of-Things, real-time processing of big data has become a core operation in various business areas. It is of paramount importance that big-data analyses are executed timely with specified accuracy guarantees. However, workloads in the wild are highly bursty with skewed contents and often present the conundrum of meeting latency and accuracy requirements simultaneously. In this paper we propose AccStream, which selectively samples and processes data tuples and blocks on emerging batch streaming platforms with a special focus on analysis of aggregation, e.g., counts, and top-k. AccStream dynamically learns the latency model of analysis jobs via on-line probing technique and employs sample theory to determine the lower limit of data so as to fulfill given accuracy targets. A unique feature of AccStream ensuring strong latency-accuracy fulfillment even under conflicts is the hybrid windowing that trades off data freshness via a combination of tumbling and rolling windows. We evaluate the prototype of AccStream on Spark Streaming, analyzing Twitter data. Our extensive results confirm that AccStream is able to achieve the latency and accuracy target against a wide range of conditions, i.e., slow and fast dynamic load intensities and content skewnesses, even when facing conflicting latency and accuracy targets. All in all, the effectiveness of AccStream in delivering timely, accurate, and (partial) fresh streaming analytics lies in shedding the adequate amount of input data at the right time and place.**

I. Introduction

The prevalence of social media and Internet-of-Things (IoT) devices catalyzes the emergence of big-data analytics which aims to discover societal, environmental, and business insights from large volumes of streaming data in a real-time fashion [1]. It is of paramount importance that such analytics are conducted with high accuracy and within short response times while they are actionable, e.g., using Twitter data to detect earthquakes [2]. The unique three V characteristics of big data, i.e., volume, velocity, and variety, increase by many folds the challenges of delivering timely and correct analysis. Recent developments in batch systems, such as Hadoop [3] and Spark [4], show a strong promise to deal with the high volume of big data problems, while the conventional streaming systems, such as System S [5] and Storm [6], are capable of processing data at high velocity. Moreover, big data in the wild exhibits not only strong temporal variability in intensity [7] but also in content skewness [8], e.g., the keywords popularity is volatile. Solutions that can address all three Vs dynamically are critical for real-time big data analytics.

Providing abundant computational and storage resources is a common solution that can be overly expensive or even insufficient to meet the latency target, as peak demands are often two to three orders higher than the average load [9], [10] and happen in transient bursts. Overload management by dropping loads [11], [7], [12] exceeding the capacity is thus often adopted to ensure the latency target at the potential risk of losing analysis accuracy. The recent advances in approximate computing argue for the need to develop solutions that strike tradeoffs among accuracy, latency, and resource requirements, being in hardware [13], programming languages [14], batch platforms [15], [16], streaming systems [7], and databases [17]. However, while the prior art can effectively fulfill the latency or accuracy target, little is known on addressing the conflicts between these two objectives.

In this paper, we present AccStream an accuracy-aware overload management that selectively drops/samples input data for batch streaming systems. AccStream aims to fulfill the average latency and accuracy targets simultaneously for real-time analysis against different dynamics of load intensities and content skewnesses. AccStream determines the data processing upper bound that results into the response time less than the latency target via a linear latency model. This model is learned via online system probing to accommodate dynamic loads. AccStream computes the data processing lower bound that satisfies the accuracy requirement for single (e.g., counts) or ensemble (e.g., top-k) estimates. In case of conflicts between the latency and accuracy targets, i.e., the lower bound is higher than the upper bound, AccStream trades off data freshness by using aggregated historical data for faster processing and higher statistical confidence. To such an end, AccStream adopts the concept of hybrid windowing which uses tumbling windows in non-conflict scenarios and switches to rolling windows to resolve conflicts. We particularly focus on two types of analysis, i.e., aggregation (single estimates) and top-k (ensemble estimates), and use Spark streaming for

the proof-of-concept prototype of AccStream. For top-k analysis, we specifically devise a searching scheme that computes the minimum data required to have non-overlapped confidence intervals (CIs) of all k items.

To achieve users' specified targets, AccStream is composed of three key components: controller, collector and enhanced receiver. The controller determines the optimal amount of data to sample/process based on the statistics of accuracy and latency monitored by the collectors. We modify the receiver of Spark streaming to support data sampling at the granularity of tuples and blocks. To evaluate AccStream, we use a Twitter trace amounting to 1 GB and 17M tweets to perform aggregate and top-k analyses on Spark streaming under static/dynamic load intensities and different content skewnesses. We define accuracy as the relative error, i.e., the ratio of confidence interval over the estimate. Our results show that AccStream can effectively drop data to fulfill the latency target while satisfying the accuracy target with (partially) fresh data against big-data in the wild.

In summary, we make the following contributions: (1) We develop an accuracy-aware overload management module, AccStream, for aggregation and top-k analyses; (2) AccStream combines tumbling and rolling windows to trade off data freshness for higher statistic confidence and lower accuracy. (3) The online-learning latency model enables AccStream to adapt agilely to time-varying loads and content skewnesses. (4) We implement our approach on Spark Streaming and extensively evaluate it on social media streams.

II. Problem Statement & Background

Today's users are interested in knowing different types of analytics from real-time IoT or social media data, e.g., average weather conditions across regions or Twitter hashtag rankings. To assure the quality of users' experience, processing platforms aim to provide such analyses within a given time constraint and guaranteed statistical confidence. On the one hand, the time constraint is to ensure the truthful reflection of current system state which can sometimes have short reaction windows. For example, knowing a sudden surge in the word/hashtag "earthquake" needs to be timely to enable fast intervention. However, the response time grows with the amount of data that needs to be processed. On the other hand, to fulfill high analysis accuracy, one needs to compute the estimates with higher statistic confidence and hence requires more data. The confidence interval of estimates thus is widely used as an accuracy metric. In this paper, we particularly consider the accuracy requirement by the relative error that is defined as the ratio of the confidence interval over the estimate.

Following the example of counting "earthquake", the question we address is how a data processing system can publish the count every 30 s with relative error below 5%, over fast input streams exhibiting high variability in the intensities and skewnesses. To answer such a question, one first needs to know how much data and time it takes to produce the result while fulfilling the relative error target. Based on the prediction of latency, accuracy, and loads, the processing system determines the amount of data to sample/drop/analyze to fulfill the dual objectives of latency and accuracy. Moreover, due to time varying dynamics of the load and content, an online learning procedure is deemed more effective than an offline one.

A. Spark Streaming

As the underlying processing system we here consider batch streaming systems with task-based parallelization, e.g., ones based on the MapReduce paradigm. Particularly, our reference system is Spark Streaming, a layer on top of Spark for stream processing. In the following, we first detail Spark then highlight the streaming components.

Spark: Figure 1 shows the overview of a Spark cluster with one master and multiple workers. The master coordinates the whole cluster and hosts the Spark engine and Streaming scheduler. The worker host executors for running the Spark tasks or the receivers which are additional components from Spark streaming. Each executor holds onto one or more CPU cores. The Spark engine processes jobs following the discipline of first-come, first-served (FCFS) and a MapReduce-like concept, i.e., jobs are decomposed into multiple tasks. At every job start, the Spark scheduler distributes the tasks to the executors. Each task is responsible to process a subset of data, which can be stored locally or at remote workers. In the case of Spark Streaming, all data is stored at the receivers. We explain how data blocks are generated on the fly in the Streaming layer.

Streaming Layer: Compared to Spark, Spark Streaming comprises two additional components: the *receivers* and the *streaming scheduler*. The former are hosted on the workers and the later on the master. The receivers accumulate incoming stream data into *blocks* based on fixed time intervals. The streaming scheduler further batches blocks into *jobs* based on separate fixed time intervals. The query execution has the following flow. First, data tuples are generated from the source at various rates (e.g., 1 MB/s). At each receiver, every arriving data tuple is then stored in a *tuple queue*. At regular times termed *block interval*, e.g., every 100 ms, each receiver empties the tuple queue by grouping all its data into a block. All generated blocks are stored in a *block queue*. The streaming scheduler (different from the Spark scheduler within the Spark engine) creates a job by taking all the blocks in the *block queue* every

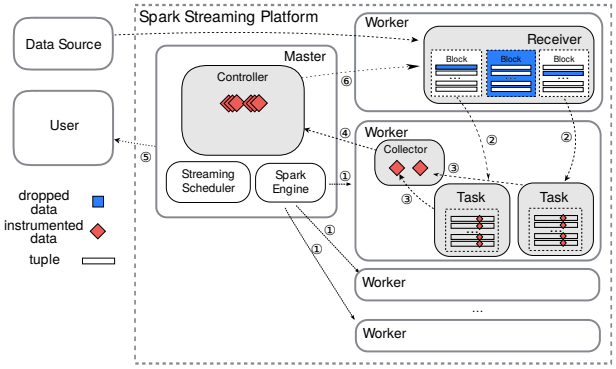

Fig. 1. Architecture of AccStream on Spark Streaming with detailed illustrations on two workers hosting receivers and executors.

job interval, e.g., 4 s, and hands the job to the Spark engine. We note that both job and block intervals are tunable parameters, the optimization of which can be found in [18].

III. AccStream Overview

AccStream is a management module that takes the latency and accuracy targets, together with load dynamics as inputs and determines how much and how to sample/process input data on Spark Streaming.

A. Architecture

AccStream relies on three components: enhanced receivers, one *collector* per Spark worker and one *controller* at the Spark master. These components are highlighted in gray in Figure 1. Compared to the default receivers, the enhanced receivers have the added ability to drop data at the block and tuple level. The collectors monitor the system and report results from each streaming job to the controller. The function of the controller is threefold. First, the controller aggregates the data received by all collectors to produce the final result. Second, the controller tunes the sampling rate(s) of the receiver(s) according to the latency and accuracy estimates. Third, the controller leverages hybrid windowing as a novel control knob to solve conflicts between latency and accuracy requirements.

Here we explains the interactions among the three components and Spark Streaming as shown in Figure 1 and leave the specifics of the controller to Section IV. To clarify these interactions, we detail the internals of two workers, one hosting a receiver and one hosting two Spark tasks, and further mark the workflow steps across one Spark Streaming job by numbers. The enhanced receivers continuously sample incoming tuples from the source and drop data at both the block and tuple level according to the sampling rates set by the controller. The instrumented (dropped) data is highlighted via red (blue) in the figure. Once every task completes processing its data block fetched from one of the receivers, the collectors record the statistics required for estimating the latency and accuracy and produce the analysis result. After receiving statistics from all the collectors, the controller computes the final result and sends it to the user. Moreover, the controller updates the sampling rate for the next job interval at the receivers.

We instrument Spark Streaming applications with AccStream so that when a task is executed, all necessary statistics can be collected at the *collector* which reports to the *controller*. All latency and accuracy parameters can be specified in the configuration file at launch time of the Spark application or adjusted at runtime for maximum flexibility. The programmatic control provides additional benefits, such as the possibility to save processing resources on the cluster during busy hours or to increase the accuracy during special events detected in the data.

B. Metrics of Interest

Analysis Latency: Here we define the analysis latency in the context of Spark Streaming as the time difference between when the job is sent to the Spark engine with all the blocks received within this job interval and when all its tasks are completed. Such a latency L includes queuing time q, *scheduling wait* δ, and job processing time. q and δ are the wait time at the FCFS queue and before the Spark scheduler computes its task graph and placement, respectively. The job processing is the time between the start of a job and the end of its last task.

Analysis Accuracy: Statistic sampling theory is widely used to obtain estimates of statistics such as the count, as computing them from the entire population is overly expensive. The accuracy of estimates is often summarized by confidence intervals, which are derived from the sampling distribution. Often, the sampling distribution is approximated as a normal distribution or obtained through non-parametric bootstrapping. We resort to the approximation of normal distribution due to the advantage of negligible computation time in computing its closed form solution.

Particularly, we let θ denote the statistics of interest and S the sample. The estimate based on the sample is termed as $\hat{\theta}(S)$. We would like to find the confidence interval as the interval covering α of the estimated sampling distribution, i.e.,

$$P([\hat{\theta}(S) - a, \hat{\theta}(S) + a]) = \alpha, \tag{1}$$

where P denotes the probability, α the confidence level, and a the error bound computed from the sampling distribution. Given the same α, the smaller the sample is, the larger the resulting a is. As an accuracy metric, we further define the estimated relative error ϵ as

$$\epsilon = \frac{a}{\hat{\theta}(S)}. \tag{2}$$

C. Enhanced Receiver

Spark Streaming sees the incoming data streams as a sequence of tuples which the receiver chops into blocks based on the block interval. Jobs thus process a set of blocks, each containing a collection of tuples. To compute the analysis, AccStream can sample/drop data either at the granularity of blocks or tuples such that both latency and accuracy targets are met. As such, we resort to two-stage sampling [19] at the receiver as it best maps to the context of Spark streaming, i.e., how to sample tuples and blocks.

We implement a two-stage sampling which uses simple random sampling at both levels. We first choose b blocks out of the total number of B blocks in a job and then choose t_i tuples out of T_i tuples in each block i. We characterize the effect of two-stage sampling on the analysis latency and accuracy via its two degenerated cases: stratified sampling, where $b = B$ which corresponds to tuple sampling only, and cluster sampling, where $t_i = T_i$ which corresponds to block sampling only. Moreover, we include a comparison of a two-stage sampling where half the data is dropped in blocks and half in tuples.

Figure 2 reports the measured latency and estimated relative errors across different sampling ratios using a word count analysis. Intuitively dropping entire blocks should provide a higher processing advantage over tuple sampling, since it avoids the overhead of creating and managing a task, as well as, of parsing the incoming data for tuple delimiters. Indeed in Figure 2, the more data is dropped the lower the latency is. Counter to intuition, block and tuple sampling achieve similar results. The main reason is that sampling and processing are decoupled both in space and time. In space, because sampling is done at the receiver whereas processing is done by the Spark tasks. In time, because the data is first sampled and stored in the block queue and then processed by the Spark engine at the next job interval. This reduces the advantage of dropping entire blocks/tasks. Finally, we observe that the results from two-stage sampling lie in between block and tuple sampling. This confirms that two-stage sampling is the generalization of tuple and block sampling. We note that in all cases we kept the number of sampled blocks high enough to avoid idling workers.

Turning to accuracy, we compare the estimated relative error based on the two-stage sampling theory [19] given a target confidence level of $\alpha = 95\%$. With the same amount of dropped data, tuple sampling provides more accurate estimates than block sampling as shown by the lower error bounds. The reason is that with block sampling there is a possibility to sample overrepresented or underrepresented blocks which can skew the results. Finally, two-stage sampling lies again in between.

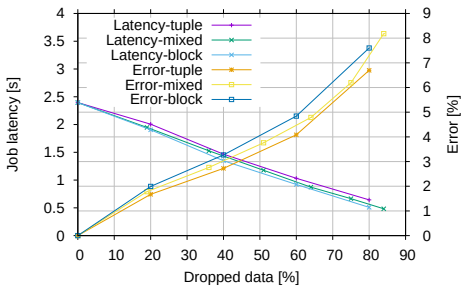

Fig. 2. Comparison of block (cluster), tuple (stratified) and mixed (2-stage) sampling.

IV. Controller

The controller is responsible to decide the optimal amount of data to drop given a dual target of analysis latency and accuracy. In this section, we describe in detail the limits on the amount of data to process based on the accuracy and latency targets, as well as how to leverage data freshness in case of contradicting requirements, i.e., using historical data to increase the accuracy. First we compute the upper and lower data limits based on the latency and accuracy targets leveraging online system probing. Then we set the sampling rates based on the tradeoff between the two data limits, solving conflicts via hybrid windowing.

A. Upper Data Limit D_u

Due to the positive correlation between admitted data and analysis latency, the latency target sets an upper limit on the amount of data to admit. Predicting the performance of MapReduce applications in general is challenging due to high fluctuation in the processing time caused by the presence of stragglers, data skew, scheduling wait, network/disk access, and other unknown reasons. A detailed performance prediction is out of the scope of this paper. Rather inspired by the latency results obtained in our data sampling characterization presented in Figure 2, we opt for a simple linear model.

Based on the definition of analysis latency, one can write L as the summation of queuing time, the scheduling delay, and job processing time. We approximate the job processing time as the product of the amount of data, D, to be processed and the processing rate, μ, defined as the time needed to process one data unit. Hence, we model the analysis latency L as follows:

$$L = q + \delta + D\,\mu. \tag{3}$$

By inverting (3) and substituting L by its target L^*, we predict at each job interval the upper limit, D_u, on the amount of data to process :

$$D_u = \frac{L^* - q - \delta}{\mu}, \tag{4}$$

where we compute q from the expected job completion time of the currently started job and estimate δ and μ through online system probing.

Online System Probing: The latency model needs the estimates of system parameters, i.e., δ, μ and data arrival rates λ. In stream processing the load is not guaranteed to be constant over time, rather the system must be able to handle time-varying arrivals. Moreover in modern cloud deployments also the system capacity can fluctuate over time either explicitly via scaling out or implicitly due to interference from neighboring workloads. Hence, we opt for continuous online probing and training.

We estimate highly dynamic metrics, i.e., the arrival rates λ, using exponentially weighted moving average that is reactive to load changes but still filter out high frequency noise. We estimate δ and μ, that have a more steady nature, via linear regression on the recent past for higher accuracy. We constantly keep track of the goodness of estimates for λ, μ, δ, and L. If we detect that fitting degrades, particularly for the analysis latency, we determine the sampling rate by switching to a more conservative control law, i.e., additive increase multiplicative decrease (AIMD), instead of following the control law specified in Eq. 3. There we increase the sampling rate by a constant whenever we meet the latency target, and we cut the rate in half whenever we violate the target. Such a control law is to help stabilize the system. Once the goodness of fit is acceptable again, we switch back to the control law of Eq. 3. The metrics are checked at the beginning of each job interval.

B. Lower Data Limit D_l

The lower data limit is determined by the accuracy target as a lower estimation error requires a higher amount of data being processed. We consider single and ensemble analyses, which have different requirements on confidence intervals. In particular we consider aggregation operators for the first, and top-k ranking for the second type, respectively.

Single Analyses: In single analyses the user specifies a relative error used as accuracy target. Here the lower data limit is determined by estimating the error bound based on the variance of the estimators used in two-stage sampling.

The estimated variance for the two-stage sum estimator [19] is:

$$\widehat{\text{Var}}(\hat{\tau}) = B\,(1-\frac{b}{B})\frac{s_b^2}{b} + \frac{B}{b}\sum_{i=1}^{b} T_i\,(1-\frac{t_i}{T_i})\frac{s_i^2}{t_i}, \quad (5)$$

where s_i^2 is the estimated variance inside block i, s_b^2 is estimated variance across blocks and T the total amount of tuples. Targeting a confidence level $100(1-\alpha)$, the error bound a is given by:

$$a = t_{n-1,1-\alpha/2}\,\sqrt{\widehat{\text{Var}}(\hat{\tau})}, \quad (6)$$

where $t_{n-1,1-\alpha/2}$ is the Student t-distribution with $n-1$ degrees of freedom. At each job start, using (5) and (6) we search over the batch $\frac{b}{B}$ and tuple $\frac{t_i}{T_i}$ sampling rates for a lower data limit D_l able to satisfy the user specified relative error target ϵ^*, i.e., $\frac{a}{\widehat{\theta(S)}} \leq \epsilon^*$.

Two-stage sampling theory provides analogous estimators for a broad-range of aggregation operators [19], Similarly, extreme value theory [20] provides estimators for extreme value operators, such as minimum and maximum.

Ensemble Analyses: In ensemble analyses the accuracy is given by the error on multiple values and their relative distances. Considering top-k ranking, the accuracy relates to the relative ordering of the k-values. The probability of reordering depends on the relative distance between neighboring elements as well as on their confidence intervals. For example, if i^{th} and $(i+1)^{th}$ adjacent elements are close with large confidence intervals, the reordering probability is high especially if the intervals overlap. On the contrary, if the two elements are distant with small confidence intervals, the reordering probability is low.

We take advantage of this special structure to derive lower limits on the amount of data to process dynamically with the aim of avoiding reordering. We enforce that two neighboring elements i^{th} and $(i+1)^{th}$ must have non-overlapping confidence intervals, i.e., assuming descending ranking:

$$\begin{aligned} \widehat{\theta_i}(S) - a_i &> \widehat{\theta_{i+1}}(S) + a_{i+1} \\ \widehat{\theta_i}(S)(1-\epsilon_i) &> \widehat{\theta_{i+1}}(S)(1+\epsilon_{i+1}), \end{aligned} \quad (7)$$

where $\hat{\theta}_i$ and ϵ_i are the estimate and relative error of element i, respectively. To be extra cautious in avoiding overlapping confidence intervals, we introduce a safety factor γ that specifies the minimum distance of every adjacent confidence intervals relative to theirs values. To such an end, we inflate each relative error by dividing it with $0 < \gamma \leq 1$ and still ensure the non-overlapping condition:

$$\widehat{\theta_i}(S)\frac{1-\epsilon_i}{\gamma} > \widehat{\theta_{i+1}}(S)\frac{1+\epsilon_{i+1}}{\gamma}. \quad (8)$$

Based on this condition and the above error bound estimates, we search for the most stringent data limit across all elements $1 \leq i \leq k$ of the top-k ranking.

C. Latency-Accuracy Tradeoff

The latency and accuracy targets provide an upper and lower limit on the amount of data to process at each

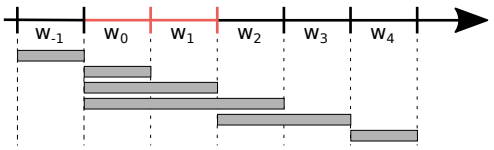

Fig. 3. Hybrid windowing: first use tumbling windows on non-conflicting intervals, then rolling windows on conflicting intervals (i.e., w_0 and w_1) and slowly switch back to the tumbling windowing. The gray bars denote the amount of data included to compute the analysis at each interval.

job interval. Unfortunately latency and accuracy follow opposite trends. Hence, we can have the following two cases: i) Accuracy provides a loose requirement with respect to latency, i.e., $D_u \geq D_l$. In this case, depending on the user preference, it is possible to choose if to prioritize a lower latency and save compute resources by setting the amount of data to process $D = D_l$ or if to prioritize a lower error bound by setting $D = D_u$. ii) Accuracy provides a strict/incompatible requirement with respect to latency, i.e., $D_u < D_l$. In this case we can satisfy either the latency or accuracy target but not both. However, to keep the system stable and avoid a latency explosion, we limit the amount of data to process to $D = D_u$ and use hybrid windowing to enhance the accuracy. In both cases, the controller updates the sampling rate at each receiver as the fraction between D and the estimated incoming data for the next job interval.

Hybrid Windowing: To solve conflicts between latency and accuracy, i.e., when we can not process enough data to reach the given accuracy target while meeting the latency target, we turn to historical analysis results to get more accurate results through hybrid windowing.

The operation of hybrid windowing is exemplified in Figure 3 over a series of job windows w_i in time. Normally the intermediate results from parallel tasks are discarded at the end of each job interval, following a tumbling window operation, see operation at w_{-1} and w_4. During periods with conflicts, see w_0 and w_1 (highlighted in red), hybrid windowing switches to a rolling window operation by increasingly using per-task results of previous jobs to obtain higher accuracy. Once the conflict is resolved, AccStream gradually reduces the amount of historical data till it reverts to the tumbling window operation. Using per-task results from multiple job intervals not only increases the sample population and thus reduces the variance of the estimates, but also reduces the computational overhead. This allows AccStream to alleviate the conflicts at the cost of staler results.

When the accuracy target is not met, we increase the size of the history windows. However too stringent accuracy targets might never be reached even with large windows. To avoid that the amount of history data grows indefinitely, we set a user tunable limit on its size. Via this limit the user decides his willingness for more accurate over more timely results.

V. Testbed Setup

We implement AccStream on Spark Streaming 1.6.2 and test it in a private testbed of 16 virtual machines (VMs). The hosting physical server is a Dell PowerEdge M620 equipped with two Intel Xeon E5-2680 CPUs at 2.7 GHz totaling 32 hyperthreaded cores, and 128 GB of RAM running Ubuntu 14.04 with kernel version 4.2 and qemu/KVM version 2.0.0 as the hypervisor. Out of the 16 VMs, 13 host Spark Streaming with 1 master and 12 workers, whereas 3 VMs host the data generators. Each VM is equipped with 1 CPU core and 8 GB of memory, except the master equipped with 4 CPU cores and 16 GB since it provides all necessary services to Spark.

Spark Streaming: Spark and Spark Streaming offer a large number of parameters to tune the system performance. In the following we summarize the parameters that differ from their default values. We configure each Spark worker with one core and 6.8 GB of memory and set the storage level to MEMORY ONLY and locality-wait to 0, to maximize the system performance. Finally, we configure the job interval to 4 s and the block interval to 100 ms such that each receiver generates about 40 blocks per streaming job.

Workload: We run our applications against real data collected from Twitter over a 1-week time period and develop a flexible in-house generator to inject the trace as a stream into the system. The generator is written in C++ and supports both fixed and variable rates. To generate different rates while keeping the data statistics coherent with the overall trace, the generator samples the trace based on fixed intervals. To overcome the limit of a single generator and stress the system under high arrival rates, we deploy three generator-receiver pairs. Moreover, to take full advantage of the system parallelism, we ensure at least one block per executor per streaming job. With nine executors and three generators we have about 120 blocks per streaming job, which allows a block sampling rate as low as 10% to account for small perturbations. If not otherwise specified, we set the latency target to 3.6 s which allows the model to have an arbitrary 10% safety margin against the job interval, and the relative error target to zero which means that the system will not drop data unless required so by the latency target.

VI. Evaluation

We adopt benchmarks in the context of Twitter streams to demonstrate the key features and effectiveness of AccStream. We first demonstrate the effectiveness of AccStream in maintaining system stability by controlling

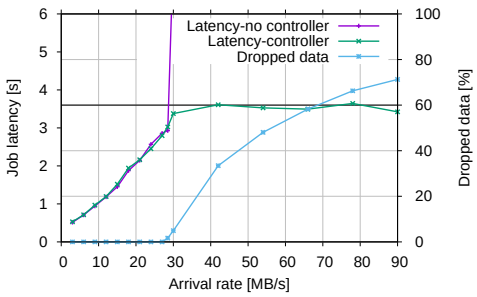

Fig. 4. Analysis latency comparison of word count with and without the controller at different arrival rates.

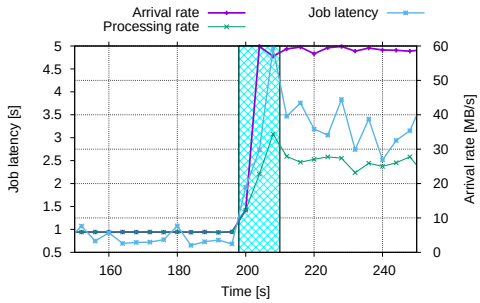

Fig. 5. AccStream applied on word count analysis that undergoes 10x step intensity change in traffic.

the job latency only, to then move on to tradeoff scenarios between job latency and analysis accuracy with and without conflicting targets. We conclude with a use case based an approximated top-k under dynamic load.

A. Latency Control

Here we focus on the effectiveness of the AccStream controller to handle overload scenarios while maintaining system stability and the latency bounded by a given target. To do so, we use counting hashtags/words in tweets under scenarios with constant and dynamic arrival rates. In these scenarios, we prioritize the analysis latency temporarily ignoring the tradeoff with accuracy.

1) Static Load: Figure 4 presents average achieved analysis latency in seconds under different constant arrival rates, with and without the controller. We see that at the beginning the latency grows with the arrival rate with no notable difference between the system without and with the controller. This shows that the overhead introduced by our controller is minimal. However, at around 28 MB/s the two system behaviors diverge. On the one hand the latency of the system without the controller starts increasing exponentially before crashing. On the other hand the system with the controller simply flattens out at around 3.6 s which is the defined latency target. This bound is maintained even at a load about 3x the maximum system capacity demonstrating the effectiveness of the latency control. Indeed this is achieved by incrementally dropping more and more data as shown by the dropped data line.

2) Dynamic Load: In the real world, both the arrival rates and system parameters might be time varying. We demonstrate the ability of the controller to adapt to such changes thanks to its continuous online system probing and conservative control by considering two dynamic loads: a step one and a slowly changing one.

Fast Load: Figure 5 shows the system response under a step change which suddenly increases the load from low (6 MB/s) to overload (60 MB/s). This sudden load increase is especially challenging for the system stability and stresses the latency control in AccStream. One can make several observations. First, it highlights the interplay between the control law in Eq. 3 and the AIMD control described in Section IV-A. The continuous updates based on the latest historical data allows the system to adapt to slow system dynamics, whereas drastic changes are handled by the more conservative AIMD control. This behavior is clearly highlighted via the figure background. Before and after the step change the control is driven by Eq. 3, whereas the AIMD takes over during the transition phase marked with a patterned background. Second, we notice that the AIMD control lasts only 1-2 job intervals more than the load transition phase. This indicates a fast response time of the controller in retraining the model. Third, we see that the combination of the two controls successfully limits the impact of even a sudden 10x load increase to one single time interval where the latency overshoots the target value ensuring the system stability.

Slow Load: Figure 6 shows the system evolution under slowly changing arrival rates. Here the load linearly increases from 20 MB/s to 58 MB/s before decreasing again to 20 MB/s over a time period of 400 s describing a triangle. We see that, the processing rate follows the arrival rate as long as it is below the system capacity (roughly 28 MB/s). Once they coincide, the controller starts to take action by dropping the amount of data in excess of the system capacity. The patterned background between the two curves highlights the dropped data. We see that the processing rate is kept around the system capacity until the arrival load drops again below it. At that point the two curves coincide again. This shows a good control of the system also under slowly varying load patterns.

B. Latency-Accuracy Tradeoff

In our previous experiments we focus on latency control and system stability. Now we consider the effects of added accuracy control using the same benchmark. In particular we consider two tradeoff scenarios: with loose and strict accuracy targets. We define an accuracy target

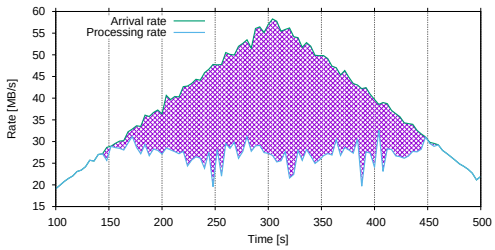

Fig. 6. AccStream applied on word count analysis that undergoes slow change in traffic intensity. The shaded area represents the dropped data.

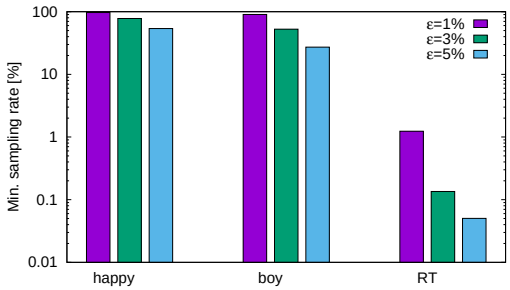

Fig. 7. The minimum amount of sampling rate required to meet different accuracy targets: counting the words "happy", "boy" and "RT".

as *loose* if the latency model allows us to sample at most $x\%$ of the data (i.e., drop $1-x\% = k\%$), however, we are able to achieve our accuracy target by sampling $< x\%$ of the data (i.e., we can drop $> k\%$). In this case it is possible to prioritize either analysis accuracy, e.g., to optimize system resources and lower the cost, or analysis latency, e.g., for faster responses. We define an accuracy target as *strict* if the latency model allows us to sample at most $x\%$ of the data (i.e., we can drop $1-x\% = k\%$), however, we must sample $> x\%$ of the data to meet our accuracy target (i.e., we can only afford to drop $< k\%$ of the data). In other words, the vanilla system may not have enough resources to process enough data within the given time to reach the given accuracy target. This leads to incompatible accuracy and latency targets which we solve via hybrid windowing.

1) Loose Target: Different hashtags/words have different statistical properties across tweets. As a consequence, the same accuracy is reached at different dropping rates. As an example, Figure 7 compares the minimum sampling rate under three different relative error targets ($\epsilon^* = 1\%$, 3% and 5%) and three different keywords. More precisely, we consider "happy", frequency $1‰$ of tweets; "boy", frequency $3‰$ of tweets; and "RT" (short for retweet), frequency 31% of tweets. One can see that the sampling limit shrinks rapidly with both the frequency of the word as well as the relative error target. A less common word such as "happy" has

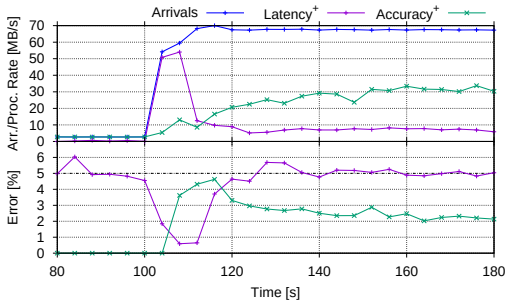

Fig. 8. The effect on processing (top) and relative error (bottom) when prioritizing accuracy or latency (marked by +) under a loose accuracy target.

higher limits on data sampling as a very common word such as "RT", e.g., 77.8% and 0.1%, respectively, for the same relative error target $\epsilon^* = 3\%$. Similar results hold for the relative error targets. A low target has higher sampling limits than a high one, for example 90.3% and 27.2% for "boy" and $\epsilon^* = 1\%$ and 5%, respectively.

Playing with these results, we demonstrate how AccStream can dynamically increase the dropping rate beyond what is required by latency target. To do so, we use the word count with "boy" as key with a relative error target of 5% and a sudden input load increase from 6MB/s to 60MB/s. Figure 8 compares the relative error and processing rate when prioritizing either accuracy or latency. In the first case, AccStream limits the processing rate to the system capacity, i.e., $\approx$ 30MB/s, based on the target latency requirement. Here the error is well below the target. In the second case, AccStream targets at maintaining the error below the set target while processing as few data as possible. Indeed, the processing rate is well below the system capacity and even slightly lower than the low input load, while the error is bounded by the target except during the load transition when the control delay shortly overshoots the required processing rate lowering the error.

2) Strict Target: Contrary to our previous scenario, users may expect a strict accuracy target which is incompatible with the targeted latency bound. To solve this problem in AccStream, we introduce a third dimension, i.e., the timeliness of the data via hybrid windowing. AccStream trades off the timeliness of data for better accuracy by increasing the time window over which the result is computed. For example, at every job interval instead of providing a result over the previous job interval, AccStream will provide a result over the previous three job intervals. This increases the size of the sample and the accuracy of the result. Since the data accuracy also depends on the underlying time-varying data statistics, this process is dynamic over time.

Figure 9 shows the case where AccStream trades the

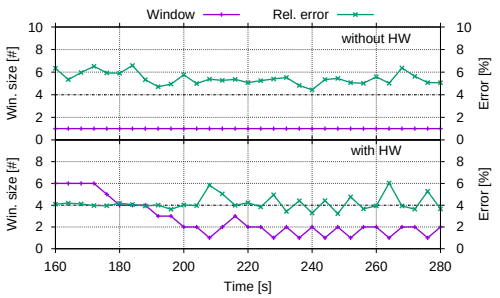

Fig. 9. Relative errors and window sizes with and without hybrid windowing (HW): counting "boy" with a error target of 4%.

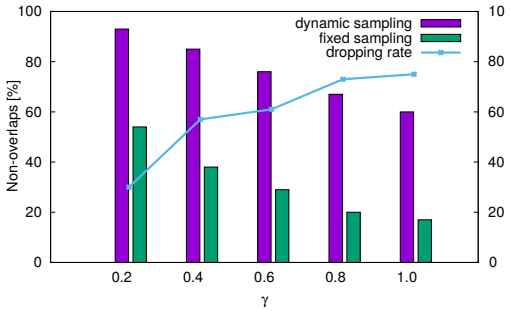

Fig. 10. The percentage of dropping rate and non-overlaps between the i^{th} and $(i+1)^{th}$ elements with approximate top-k, with respect to different values of relative distances γ.

results timeliness for better accuracy using the word count benchmark under high load, i.e., 60MB/s, and a relative error target of 4%. To show the effect of hybrid windowing, we report the window size in terms of job intervals. If the window size is one, only the per task results from the current job are used, whereas any number above one indicates the amount of historical data used. The top plot shows the scenario with hybrid windowing disabled. We see that AccStream is not able to satisfy the error target. On the contrary, with hybrid windowing enabled the achieved relative error oscillates around the target, see the bottom plot. This is at the cost of less fresh data due to the use of historical results as indicated by the window size greater than one. Moreover, we see how the window size is dynamically controlled over time to provide the best tradeoff between timeliness and accuracy.

C. Hashtag Ranking

We conclude this section with use case involving an ensemble estimate: finding the most popular hashtags in Twitter streams under dynamic load. This is a perfect match for our approximate top-k ranking. The approximate top-k controls the accuracy targets, and hence the data dropping rate, programmatically based on the distance between the i^{th} and $(i+1)^{th}$ element so that their confidence intervals do not overlap to avoid reordering. The goal here is twofold: demonstrate our approximate top-k and evaluate how the controller handles a more complex scenario which includes both dynamic loads and time-varying accuracy targets.

To evaluate this scenario we use as figure of merit the number of streaming jobs with no overlaps between the i^{th} and $(i+1)^{th}$ element. Figure 10 shows this number in percent under different safety scaling factors γ and the same dynamic input load. As a comparison, we include a baseline algorithm using a fixed data dropping rate equal to the average dropping rate used by our approximated top-k. Clearly, our approximation top-k outperforms the baseline across all scenarios. For example with $\gamma = 0.4$, our top-k approximation dropped on average 57% of the input data while providing a clear ranking in 87% of the times. On the contrary, the baseline in the same scenario obtains clear rankings only 38% of the time. This justifies the AccStream module which allows identifying the best moments to drop data while avoiding ambiguous rankings. Naturally the lower the scaling factor, the higher the margin and the clearer the rankings, but at lower data dropping rates and hence higher processing rates.

VII. Related Work

Optimizing latency is a long-standing challenge for real-time processing systems. A great number of control strategies, including resource scaling [21], program optimization [22], scheduling [23], and data shedding [11], [7], have been applied to cope with persistent and transient overloads, and thus a detailed enumeration is not feasible due to the space limit.

Admission control by dropping data according to the capacity can effectively mitigate transient overloads, either in a probabilistic random fashion [24], [25], [12] or in a semantic-aware fashion [26], [7], [27]. The metrics of merit here center on throughput [27], profit of the system [24], or utility functions [25], which measure the importance of certain data to the quality of analysis and implicitly consider the accuracy. Perez [28] and Xia et. [21] proposed analytical framework to predict the latency for distributed streaming systems and argued for the optimality via theoretical analysis. THEMIS [7] proposed to use source information content to maintain the fairness in federated streaming systems, when distributively dropping tuples in the entire operator plans.

While the aforementioned studies make significant progress in shedding policies on streaming processing systems based on operators, the tradeoff between analysis accuracy and the latency is not explicitly explored. The central theme of approximate computing is to guard the metrics of interest for users, and systems such as latency, energy efficiency at the cost of accuracy.

BlinkDB [17], an approximate query processing framework, provides accuracy guards in short response times by leveraging statistical sampling theory to choose the inputs and Spark [4]. ApproxHadoop [15] develops a two-stage sampling strategy for Hadoop [3] by either dropping the tasks or amount of data per tasks, so as to minimize the overhead of data accessing. However, these frameworks usually do not have strong time-varying workloads and require offline profiling to build the latency models to explicitly explore the tradeoff between the latency and accuracy, which is not applicable in streaming environment.

VIII. Conclusions

Proving timely and accurate analytics has become ever important in the big data era but also challenging due to the time-varying workload intensities and content skewnesses. In this paper, we propose AccStream supporting aggregation and top-k analyses in accuracy-aware Spark Streaming system. AccStream is specially designed to make a real-time tradeoff among accuracy, latency, available processing capacity, user preference, and freshness of analysis, via on-line learning latency and accuracy models. AccStream features on hybrid window that enables strong fulfillment on latency and accuracy targets by slightly sacrificing the data freshness for fast computation and higher statistical confidence. Using extensive evaluation on Twitter tweets, we show that AccStream can strongly fulfill the targets provided by users, especially encountering the overloads and experiencing conflicting latency and accuracy target. Our results confirm that AccStream is not only able to shed the adequate amount of data with the right timing but also at the right place, thanks to the exploration on approximating relative ranks. In our ongoing research, we are extending AccStream for different types of analyses as well as other types of batch streaming systems.

IX. Acknowledgements

The research presented in this paper has been supported by the Swiss National Science Foundation National Research Programme "Big Data" (NRP 75) (project 407540_167266).

References

[1] T. Sakaki, M. Okazaki, and Y. Matsuo, "Tweet Analysis for Real-Time Event Detection and Earthquake Reporting System Development," *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 4, pp. 919–931, 2013.

[2] "How the USGS uses Twitter data to track earthquakes," https://blog.twitter.com/2015/usgs-twitter-data-earthquake-detection.

[3] "Apache Hadoop Distributed File System," http://hadoop.apache.org/, Accessed: 2016-05-20.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *HotCloud*, 2010, pp. 10–10.

[5] "IBM InfoSphere Streams," http://www-03.ibm.com/software/products/en/infosphere-streams/, Accessed: 2016-05-20.

[6] "Apache Storm," http://storm.apache.org/, Accessed: 2016-05-20.

[7] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch, "THEMIS: Fairness in Federated Stream Processing under Overload," in *SIGMOD*, 2016.

[8] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load Balancing in MapReduce Based on Scalable Cardinality Estimates," in *ICDE*, 2012, pp. 522–533.

[9] R. Krikorian, "New Tweets per second record, and how!" https://blog.twitter.com/2013/new-tweets-per-second-record-and-how, Accessed: 2016-05-20.

[10] "Slashdot Effect," https://en.wikipedia.org/wiki/Slashdot_effect, Accessed: 2016-05-20.

[11] N. Tatbul and S. B. Zdonik, "Window-Aware Load Shedding for Aggregation Queries over Data Streams," in *VLDB*, 2006, pp. 799–810.

[12] R. Birke, M. Björkqvist, E. Kalyvianaki, and L. Y. Chen, "Meeting Latency Target in Transient Burst: A Case on Spark Streaming," in *IEEE IC2E*, 2017, pp. 149–158.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012, pp. 301–312.

[14] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: reliability- and accuracy-aware optimization of approximate computational kernels," in *OOPSLA*, 2014, pp. 309–328.

[15] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing Approximations to MapReduce Frameworks," in *ASPLOS*, 2015, pp. 383–397.

[16] Y. Ying, R. Birke, C. Wang, L. Y. Chen, and N. Gautam, "Optimizing Energy, Locality and Priority in a MapReduce Cluster," in *ICAC*, 2015, pp. 21–30.

[17] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *Eurosys*, 2013, pp. 29–42.

[18] T. Das, Y. Zhong, I. Stoica, and S. Shenker, "Adaptive Stream Processing using Dynamic Batch Sizing," in *SoCC*, 2014, pp. 16:1–16:13.

[19] S. Lohr, *Sampling: Design and Analysis*. Cengage Learning, 2009.

[20] S. Coles, J. Bawa, L. Trenner, and P. Dorazio, *An introduction to statistical modeling of extreme values*. Springer, 2001, vol. 208.

[21] C. H. Xia, D. F. Towsley, and C. Zhang, "Distributed Resource Management and Admission Control of Stream Processing Systems with Max Utility," in *ICDCS*, 2007, pp. 68–68.

[22] M. Samadi, D. A. Jamshidi, J. Lee, and S. A. Mahlke, "Paraprox: pattern-based approximation for data parallel applications," in *ASPLOS*, 2014, pp. 35–50.

[23] R. Birke, E. Kalyvianaki, W. Binder, M. Schmatz, and L. Y. Chen, "Dynamic block sizing for data stream processing systems," in *IC2E*, 2016.

[24] L. A. Moakar, P. K. Chrysanthis, C. Chung, S. Guirguis, A. Labrinidis, P. Neophytou, and K. Pruhs, "Admission control mechanisms for continuous queries in the cloud," in *ICDE*, 2010, pp. 409–412.

[25] C. Balkesen, M. T. Ozsu, and N. Tatbul, "Adaptive Input Admission and Management for Parallel Stream Processing," in *DEBS*, 2013, pp. 15–26.

[26] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami, "Selectivity and Cost Estimation for Joins Based on Random Sampling," *J. Comput. Syst. Sci.*, vol. 52, no. 3, pp. 550–569, 1996.

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

[28] J. F. Perez, R. Birke, and L. Y. Chen, "On the Latency-Accuracy Tradeoff in Approximate MapReduce Jobs," in *IEEE INFOCOM*, 2017.