An Online Approximate Stream Processing Framework with Customized Error Control

Xiaohui Wei, Yuanyuan Liu, Xingwang Wang✉ , Shang Gao
College of Computer Science & Technology, Jilin University, China

Abstract—**In online approximate stream processing, customers generally submit their requests with some specific quality requirements (*e.g.* maximum error). This raises a critical problem that online quality control is necessary to meet customized requirements. Since continuous arriving data needs to be processed immediately, it brings the difficulty of acquiring knowledge which significantly affects the efficiency of sampling. Hence, it's more challenging to ensure a prescribed level of quality without knowledge about data. In this paper, we present an adaptive approximate processing framework for online stream applications to address the challenges mentioned above. Specially, we first design a new data knowledge learning scheme to stratify the arriving stream data. Then, based on the online learning results, we propose a dynamic sampling strategy with the consideration of the stream rate. Finally, we further present a double-check error control mechanism to manage the output quality. Experiments with real world datasets show that the proposed approximate framework is not only applicable to different data distributions, but also provides a customized error control.**

I. Introduction

Online stream processing can transform continuous raw data into valuable information, which is widely applied to various fields including network traffic monitor [1], sensor-based measurement networks [2]. Among them, data continuously arrives and users are concerned about real-time analysis results, such as querying within a specific time period.

However, because of the considerable data volume and quick arriving rate, processing big data streams is still challenging, resource-consuming although it can be processed by general distributed stream processing systems (DSPS) [3]. In this case, approximate computing as an effective solution paradigm, can be applied to obtain results quickly while ensuring the specified level of accuracy [4]. Combined with distributed processing models (*e.g.* Spark [3]), approximate computing is attracting more attention to achieve low latency and efficient resource utilization.

The common approximate technique applied in large-scale analysis is sampling. Sampling-based approaches have been extensively studied for aggregation queries [4]. The efficient sampling method can process large datasets to get a smaller sample dataset on which queries are executed to return a customer-satisfied result. The customer requirements can refer to either a desired error bound, or query response time [5], and the efficient sampling strategy should be configurable so as to satisfy different user requirements.

✉ : Corresponding author, Email: xw.wang1990@gmail.com

General sampling methods can be executed based on the known or predictable data knowledge (*e.g.* distribution, maximum, minimum). Most existing works assume the characteristics of arriving data can be obtained from historical logs [6]. The preprocessing operation is used to make preparation for the sampling operation. However, these strong assumptions may lead to inconsistent prediction that produces ineffective samples for online processing. Compared with stored data sets, it's more difficult to make effective cognition for real-time data stream processing. Since online stream data continuously arrives without being stored and the data knowledge is unknown in advance. Different from assuming a priori knowledge in existing works, we develop an online data cognition in order to better adapt to the dynamic change of stream data.

Besides, approximate computing can not only improve processing performance, but also ensure the output meets a prescribed level of quality. The quality requirement is generally specified by users, such as maximum or average error, response time, etc. Especially in real-time stream processing where data is not being stored, it's more necessary to check the output to assure its quality so that the unsatisfactory results can be timely corrected.

Currently the existing studies tend to provide an error guarantee [6]. In more detail, when sampling a dataset, there will return an approximated result with a theoretical error bound within the interval where the true value falls with a high possibility (known as *confidence*). However, owing to the probability of sampling, it's possible that approximate computing produces unacceptable errors, which will affect the final output quality and reduce customer satisfaction [5], [7]. Hence, the online quality control for processing stream data is necessary, which has not been considered in existing works. To make effective output quality, we need to design online strategies to control approximate results for customized requirements, which is the goal of our work.

In conclusion, there are two critical problems to address for online stream processing: online data cognition and quality control. To tackle these problems, we propose a general *approximate stream processing framework* in this paper. The framework is composed of three main components: a new data learning scheme, a dynamic sampling strategy and a customized error control mechanism. Online data learning scheme is designed to acquire the overall data distribution, which will be updated in realtime. Then the dynamic sampling strategy makes sampling with the consideration of the fluctuating stream rates. Without requiring application-specific

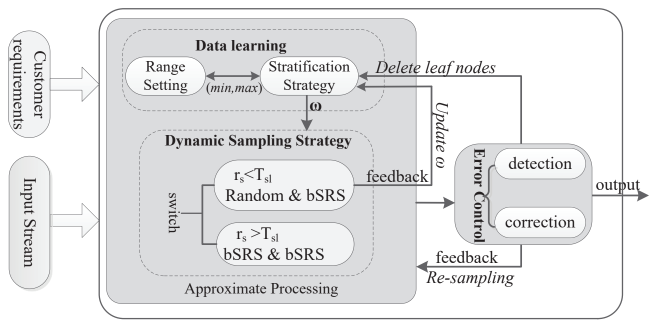

Figure 1: A general approximate processing framework over online data stream.

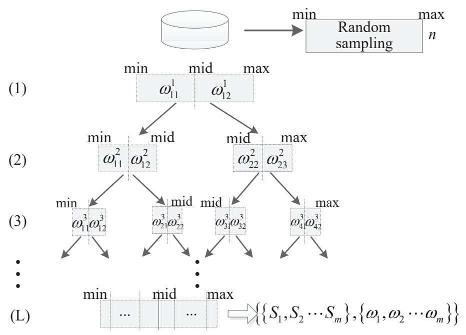
min
max
Random sampling
n
(1)
(2)
(3)
(L)
mid
$\{\{S_1,S_2\cdots S_m\},\{\omega_1,\omega_2\cdots\omega_m\}\}$

Figure 2: The process of stratification with the binary tree.

approximate algorithms, the sampling module can execute a self-adjusting computation. Furthermore, the error control module leverages an output-based method to detect output and find errors that need to be corrected. With these modules, our designed approximate framework can effectively process online stream data and adaptively meet different customized requirements.

Our **contributions** of this paper are as follows:

(1) We propose a new online data learning scheme with a triggered weight update strategy. The scheme can analyze the constantly arriving data to make a weighted stratification.

(2) We present a dynamic sampling strategy that switches to different sampling methods based on the varying stream rates.

(3) We design a customized error control mechanism, which provides a feedback mechanism to detect and timely correct large errors.

(4) Experiments with real-world datasets are conducted to validate the effectiveness of our proposed approximate processing framework.

II. Design of Approximate Module

In this section, we present the online approximate processing module including online data learning and dynamic sampling strategy.

A. Online Data Learning

Firstly, we design a stratification strategy to make a new data cognition. The online data learning adopts a divide-and-conquer method. Considering the data distribution is a critical factor for online stream applications, the stratification strategy progressively divides the stream according to the value range of data. Then at the end of partition, each sub-range corresponds to a weight and is converged to the final learning result.

1) Data Range Setting: Before stratification, the value range of stream data (min, max) need be set. To online get the value range, we utilize the idea of invalid timer that is applied in network routing protocol to set the value range [8]. For real data stream, the varying stream causes that the value range may change over time. To adapt the online stream data, we first present a dynamic range update scheme based on a timer.

To explain the process, we use the minimize value as an example. There are two parameters: the setting minimum min and the observed minimum min_o. Initially, a timer is empirically set and the value range can be set randomly. When data arrives and the timer isn't expired, we compare the value of min and the update observed minimum min_o as follows: if detecting $min_o = min$, the timer is reset; if $min_o < min$, then we update min with min_o and reset the timer. Moreover, when the timer is expired, we also need to reset the timer and update min with min_o that represents the recent minimum value. Through the control of the timer, the value range can be online gotten and dynamically updated.

2) Stratification Strategy: With the current value range, the method needs to divide the arrival data set into two or more strata while each stratum selects different and appropriate weights. Figure 2 shows we use a binary tree structure to express the process of weight selection. Firstly, an item set derived by random sampling scheme can be obtained, which is the reference sample in the following stratification phase. For random sampling, the sampling weights from the minimum to maximum at this phase are same. Assume the average estimated by random sampling is $\hat{v}_0$.

In the first level, data items are divided into two strata according to their value ranges, respectively (min, mid) and (mid, max). With the value of $\hat{v}_0$, we can firstly analyze the weights of these two strata, denoted as ω_{11}^1 and ω_{12}^1. After stratification, the average value of each stratum can be obtained based on data items from each stratum, denoted as $\hat{v}_{11}^1$ and $\hat{v}_{12}^1$. Then the estimated average value of the first level can be computed:

$$\hat{v}_1^1 = \frac{1}{2}\hat{v}_{11}^1 + \frac{1}{2}\hat{v}_{12}^1 \tag{1}$$

Compared with the initial value $\hat{v}_0$, the weight of each sub-range (ω_{11}^1 or ω_{12}^1) can be modified. The modified weight is based on the definition that the sampling weight is the reciprocal of the inclusion probability: $\omega_i = \frac{1}{\pi_i}$, where π_i is the probability that unit i is included in the sample [9]. Denote β as the proportion tuning parameter and we utilize β to adjust the values of ω_{11}^1 and ω_{12}^1. Through comparing $\hat{v}_{11}^1$ and $\hat{v}_{11}^2$, there are:

(1). If $\hat{v}_0 \geq \hat{v}_1^1$, set $(\frac{1}{2}+\beta)\hat{v}_{11}^1 + (\frac{1}{2}-\beta)\hat{v}_{12}^1 = \hat{v}_0$ where $\omega_{11}^1 = \frac{1}{\frac{1}{2}+\beta}$ and $\omega_{12}^1 = \frac{1}{\frac{1}{2}-\beta}$.

(2). If $\hat{v}_0 < \hat{v}_1^1$, set $(\frac{1}{2} - \beta)\hat{v}_{11}^1 + (\frac{1}{2} + \beta)\hat{v}_{12}^1 = \hat{v}_0$ where $\omega_{11}^1 = \frac{1}{\frac{1}{2}-\beta}$ and $\omega_{12}^1 = \frac{1}{\frac{1}{2}+\beta}$.

Two conditions (1) and (2) correspond to different proportions of two sub-ranges. Hence, the weight values at these two strata can be modified as the above rules in the first phase. Then as shown in Figure 2, each stratified value range will be further divided to two child nodes with smaller value ranges in the next level.

For each child node, a smaller stratum will get a new weight value using the same method as described in the first level. Through comparing the approximate result with the result generated by random sampling, the scheme modifies the original stratified weights. For instance, the first step is to adjust the weights ω_{11}^1 and ω_{12}^1 based on the values of $\hat{v}_{11}^1$, $\hat{v}_{12}^1$ and $\hat{v}_0$. The modified weights will be set as the reference weights of the next stratified sampling until the partition ends. For better description, let the tree height of stratification be L that depends on the difference between the maximum and minimum value of the sub-range. In the weight computing phase, a weight learning threshold T_ω is also needed to end the process of weight modification. Eventually, at each phase the weights are modified as stream data constantly arriving.

In the end of stratification, our proposed method partitions the value range of the whole data items into m strata, $S_i = [a_i, b_i], i = 1, \cdots, m$, which corresponds to the leaf nodes of the tree, and each stratum owns a weight, ω_i. With these stratified weights, a sampling scheme can be constructed to process the real-time data stream. When the value range is changed over time, we can make new stratification based on the updated range to generate more representative samples.

B. Dynamic Sampling Strategy

As mentioned above, the dynamic approximate strategy is closely related to the varying stream rate. Firstly we denote T_{sl} as the low threshold of the data arrival rate, and the system can set the value of T_{sl} according to its processing capacity. The approximate processing module shown in Figure 1 describes a complete execution for online data stream. The current stream rate is used to trigger the switch of approximate methods.

Instead of using only one sampling method, the strategy switches the sampling schemes with the change of data arrival rate. The switching of sampling methods aims to present a feedback for weight adjustment. Here a stratified reservoir sampling algorithm (bSRS) described in Algorithm 1 can be seen as a basic sampling method for each processing window. Algorithm 1 leverages a hash mapping method to stratify the arriving stream. Based on the weights gotten from stratification strategy, it allocates the sample sizes for each stratum and then fills each sample set using a conventional reservoir sampling (CRS) [3]. When the sub-sample set is full, we use a probability $\frac{n_i}{|S_i|}$ to accept or reject the item and it is consistent in each stratum to ensure equal inclusion probability. Then if accepted, we replace a randomly selected item from the sub-sample set with the arriving item.

With the basic sampling algorithm, the dynamic sampling strategy (DSS) is described as follows. Assume the current stream rate is r_s. If $r_s < T_{sl}$, we select to perform both the random sampling and bSRS algorithm; otherwise, if $r_s \geq T_{sl}$, two stratified sampling methods can be executed in parallel. Executing sampling twice, on one hand, can give a feedback to update weights for the stratification phase. On the other hand, the result comparison between these two methods can be utilized to detect the probability of error occurrence, which will be described in the next section.

Algorithm 1 A Basic Stratified Reservoir Sampling (**bSRS**)

```
Require: a real-time data stream, sample size $n$, weight set $\{\omega_i\}$
1: $sample \leftarrow \varnothing$;
2: if the set $\{\omega_i\}$ is updated then
3:     Update the input $\omega_i$;
4: end if
5: for each current processing window do
6:     Compute the sub-sample size $n_i$ in each stratum $S_i$ according to input $\omega_i$ and $n$;
7:     for the arriving item belonging to stratum $S_i$ do
8:         if ($|sample[i]| < n_i$) then
9:             $sample[i]$.add(item);
10:        else
11:            $p \leftarrow \frac{n_i}{|S_i|}$;
12:            $sample[i]$.replace(a random item, $p$);
13:        end if
14:    end for
15: end for
```

III. Customized Error Control Mechanism

Although these approximate techniques can provide significant performance gains, it's still difficult and even expensive to monitor the output quality, especially for online stream data. An exact bootstrap for accuracy estimation needs $\binom{2n-1}{n-1}$ resamples where n is the sample size [7].

Currently, most researches concentrated on how to obtain effective approximated results with bounded errors. In theory, the bounded error is given under the condition that the exact result with at least δ-probability falls in the confidence interval. Owing to the probability of sampling, it may exists the situation that the output error is too large to provide satisfactory results. Although approximate methods provide error bound guarantee, sometimes the accuracy requirement of output results may still be unsatisfied from customers' view. Thus, in order not to affect the final output quality, we need to propose solutions to reduce these unacceptable results. With different requirements, we present a customized error monitoring strategy to detect and correct the approximate outputs.

Error detection. To evaluate and manage the output quality, it's important to ensure the error monitor model has low overhead. With the requirement, we leverage two samples to evaluate the quality of approximate output. Relative to the total dataset, it's light-weight to make comparison between two samples.

Assume the above approximation methods simultaneously produce two samples, denoted as S_1 and S_2, respectively. Then we compute two estimated AVG values, $\hat{\hat{v}}_1$ and $\hat{\hat{v}}_2$. According

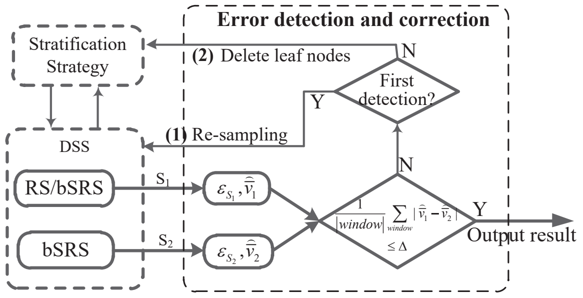
Error detection and correction
Stratification Strategy
(2) Delete leaf nodes
N
First detection?
Y
DSS
(1) Re-sampling
N
RS/bSRS
S1
$\varepsilon_{S_1}, \hat{\bar{v}}_1$
$\frac{1}{|window|}\sum_{window}|\hat{\bar{v}}_1 - \hat{\bar{v}}_2| \le \Delta$
Y
Output result
bSRS
S2
$\varepsilon_{S_2}, \hat{\bar{v}}_2$

Figure 3: The error detection and correction mechanism.

to the estimation theory of error bounds, these two AVG values are defined as:

$$\left|\hat{\bar{v}}_1 - \bar{v}\right| \le \varepsilon_{S_1} \tag{2}$$

Similarly, it's also established for $\hat{\bar{v}}_2$ with ε_{S_2}. $\bar{v}$ is the exact AVG value and ε_{S_1}, ε_{S_2} are the corresponding error bounds about samples S_1, S_2.

Therefore, in theory, the difference between two estimated values should satisfy the inequality:

$$\left|\hat{\bar{v}}_1 - \hat{\bar{v}}_2\right| \le \varepsilon_{S_1} + \varepsilon_{S_2} = \Delta \tag{3}$$

For simplicity, let $\Delta = \varepsilon_{S_1} + \varepsilon_{S_2}$ that represents the error bound obtained from user requirements. We use the value of Δ as a criterion to estimate the probability of a large error. For the maximum error constraint, if the comparison result does not satisfy Eq. (3), then the detection module will report an error. For the average error, the module will compute the average of multiple comparison results. We assume the sampling phase generates samples with the same error-bound, denoted as $\varepsilon_{S_1} = \varepsilon_{S_2} = \varepsilon_S$.

Error correction. If the error detection model detects unacceptable errors, we simply utilize the re-sampling method to correct the output. Since it is feasible to re-sample data of the current processing window, the output quality can be improved by timely correcting unsatisfactory results.

Error control mechanism. Next we propose a scheme for the output-based error detection and correction as shown in Figure 3. Our proposed approximate schema above first generates two samples for comparison, and then outputs the computed values by reference to their estimated error bounds. The computed results and corresponding error bounds are the inputs of error detection. When considering the maximum error, each time we detect the current window where $|window| = 1$. If the difference of two estimated values is larger than Δ, we can assure that at least one of the two samples produces a large error. For the average error, we need to compute the average value of comparison results generated by multiple windows during the given time period. Once there are unsatisfactory results, the error detection module will give a feedback to the Dynamic Sampling Switch (*DSS*) module. The feedback notifies to re-sample data in the current stream processing window to make correction.

The process of double-check error detection aims to provide quality-assurance output. When monitoring a large error, we will judge whether it is first detected. If so, the decision is to re-sample data. Otherwise, the module will feed back to the stratification strategy for weight adjustment. Because we think this case cannot be well corrected only by re-sampling, and the sampling weight also need to be improved to generate more accurate results. Before re-sampling, we delete the leaf nodes in the stratification binary tree and go up to select their father nodes as the reference of sampling weights. The new stratification can be implemented by the weight update strategy discussed in the next section.

Taking both re-sampling and weight adjustment into account, we can assure the output quality with a higher probability. As previously mentioned, executing two sampling schemes concurrently aims to update weights and compute the value of Δ. Then the results of error detection in turn are beneficial to update weights for better approximate outputs. These proposed modules correlate to each other and the whole approximated processing framework will be well trained so that it can be used to directly process subsequent real-time stream data.

Then, we theoretically analyze the improvement of output quality when launching the error correction mechanism. Denote the confidence values of the two samples as δ_1, δ_2, and assume that $\delta_1 = \delta_2 = \delta$.

Theorem 1. *With the error control mechanisms, the probability that the approximate error is within the given error bound at the final output increases at least $\delta(1-\delta)$.*

Proof. Given the specified confidence δ, the possibility that two sampling results both have large errors is $(1-\delta)^2$. There is at least $1-(1-\delta)^2$ probability to ensure that the approximate error is within $\pm\varepsilon_S$. Therefore, the proposed detection method can improve at least $1-(1-\delta)^2-\delta = \delta(1-\delta)$. $\square$

For instance, assume the specified confidence $\delta = 90\%$. With the quality monitoring mechanism, theoretically the confidence will be raised from 90% to 99%. In practice, there are two situations that the error detection model cannot find when two estimated results are both larger or smaller than the exact value (*e.g.* $\left|\hat{\bar{v}}_1 - \hat{\bar{v}}_2\right| \le \Delta$ but $\hat{\bar{v}}_1 - \bar{v} > \varepsilon_{S_1}$, $\hat{\bar{v}}_2 - \bar{v} > \varepsilon_{S_2}$). However, the probability of occurrence of the above case is low and it's much less than $(1-\delta)^2$.

IV. Improvement

For better optimizing our designed approximate framework, we additionally present some improvements from points of weight update and stream evolution.

A. Triggered Maintenance of Stratification Weight

The initial computed weights need be dynamically adjusted so as to better adapt to the characteristics of stream data. Owing to the continuously arriving data, the knowledge obtained from input data is gradually accumulated and updated. With more data knowledge, the designed sampling strategy provides a feedback to adjust the initial weights for better sampling.

Here, we consider the weight maintenance as a *triggered update operation*. Considering both computation load and approximate quality, there are two situations that can trigger a weight update operation:

(1) When the data arrival rate is below the specified rate threshold T_{sl} that means relatively small computation overhead, it can trigger to adjust weights of each stratum. The sampling weights can be maintained at this stage.

(2) Besides, when the output detection module judges that the approximated results occur a large error, it can trigger to adjust weights to improve the accuracy of sampling results.

Here we assume that the data distribution tends to stability. Thus, the sampling weight can be updated but not frequently produce a large change. To ensure low update overhead as possible, we do not need to re-calculate all weights. In section II-A, our proposed stratification strategy is based on the binary tree structure, and the final weights are computed according to the parent nodes. Thus, the weight can be updated by their parent or ancestor nodes of the upper levels. With the above trigger conditions, the weight update algorithm (TWU) is described as follow:

Algorithm 2 Triggered Weight Update Algorithm (**TWU**)

```
1: Receive the feedback from the sampling results;
2: Set the current level $L_c = L - 1$;
3: Based on the stratification tree structure, allocate the weights for
   stratified sampling from the $L_c$ level.
4: Compare results with random and stratified sampling methods.
5: if the weights of $L_c$ level need to be updated then
6:     Backtrack to the upper level;
7:     $L_c = L_c - 1$
8:     goto line 3;
9: end if
10: Update the weights from $L_c$ to $L$ levels based on the weight
    computation method.
11: Return the updated weights to the DSS module.
```

To faster end the loop operation of line 5, the modification on L_c in TWU at line 7 can also be set as $L_c = L_c - 2$.

B. Improvement with Stream Evolution

As the stream moves forward, the learning result of stratification tends to be stable and the corresponding output results will satisfy customer requirement with higher probabilities. At this moment, an improvement can be designed to reduce the overhead of online approximate processing.

The frequency of error control can also be reduced with the stream evolution. The process of comparing two estimated results can be set to execute in a sampling way. In DSPS, our computation model is the sliding window. Videlicet, it's not necessary to constantly monitor the quality and we can compare results to detect errors generated by partial windows other than all. Referring to the idea in Paraprox [10], it can be implemented through setting a fixed window interval *N* and checks are performed every N^{th} invocation. At other time, we do not need to execute sampling twice.

V. Performance Evaluation

In this section, we evaluate the performance of our proposed framework. Through comparing with ApproxHadoop [3], we analyze the effectiveness of online error detection mechanism.

A. Experiment results

We drive experiments with the online aggregation operation, AVG. The real dataset WikiLength, are evaluated through analyzing lengths of web pages (bytes) [11]. The large-scale dataset includes the December 2016 snapshot of Wikipedia (12.6GB), which contains more than one million English articles . We use the *Sine* function to control the arrival rate of data, which is processed in sliding windows. Here denote the statistical error of the entire dataset as ε_T and the current sliding window as ε_w.

1) Stratification learning analysis: We first implement the stratification strategy mentioned in section II-A. Figure 4 illustrates the detailed process of weight change through stream data learning. The initial weight of each strata is allocated equally and then updated with new arriving data. As described in Section II-A, a three-level binary tree is constructed and the value range of page length is partitioned into four strata. The accurate ratio of each stratum is also listed in the rightmost histogram of Figure 4. At the end of learning, the estimated weights in the first stratum is about 0.8382 compared with the accurate value 0.8405. We can see that the weight learning process gradually approaches the exact values and the final weight result is nearly consistent with them.

2) Effects of window size: As shown in Figure 5, we change the window size form 1000 to 4000 under different sampling ratios (0.05 and 0.1). When setting the sampling ratio 0.05, the approximate accuracy computed under each processing window increases by about 83% shown in Figure 5. For instance, when the window size becomes 4000, the approximate error reduces to 0.02. The reason is that the larger window sizes can obtain a more comprehensive data information, which is critical for a good learning result.

3) Effects of error control: We test the variation of ε_w with different Δ, which reflects the effect of quality control in the case of unknown exact results. In the experiments, when detecting $\left|\hat{\hat{v}}_1 - \hat{\hat{v}}_2\right| > \Delta$, the error control module will give a feedback to adjust weights as described in Figure 3. Here we fix the sampling ratio 1% and vary Δ from 30 to 200. Shown in Figure 6, when enlarging the difference of detecting large errors, the average window error increases accordingly at different window sizes. It indicates that the accuracy of results will be reduced when we relax the criterion of error detection. Experiments show that online error detection can decrease the accuracy loss and improve the output quality.

As mentioned in section IV-B, the overhead of error control can be optimized through error checking every N^{th} invocation. Next, we analyze the effects of setting different frequencies *N*(*e.g.* 1,2,4) to ε_T when varying the sampling ratio. Shown in Figure 7, the sampling error is increased when reducing the error control frequency while the corresponding sampling cost is reduced. The symbol $N : 1(1000)$ means the number of processing windows that we perform error checking is 1000. Figure 8 shows the comparison of accuracy loss with different sampling ratios when sampling with the methods of IncApprox and the addition of the error control mechanism. The results

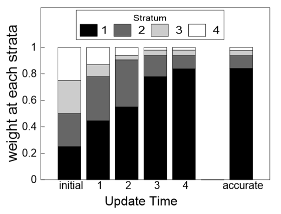

Figure 4: Weight learning result.

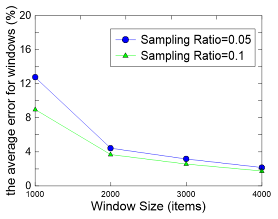

Figure 5: Effects of window size.

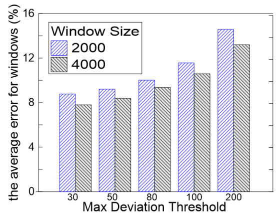

Figure 6: Effects of different threshold Δ .

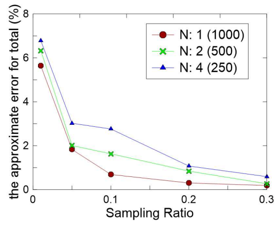

Figure 7: Effects of error control frequency N^{th}.

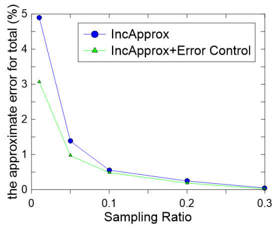

Figure 8: Comparison with IncApprox.

also indicate that the error control strategy can contribute to a better approximate result.

VI. Related Work

More and more large data researches concentrate on approximate streaming computing using sampling techniques [12]. For instance, Aggarwal *et al.* designed a temporal biased reservoir sampling [13] with the effect of time during stream data continuously arriving. Although these common sampling methods are available for streams, it's still inefficient without considering the context of stream data. In recent works, Krishnan *et al.* [3] implemented a stratified reservoir sampling algorithm based on Spark framework. The initial sampling proportion depends on the number of items seen in the current sliding window and will be adjusted periodically. For stratified stream sampling, [14] proposed two challenges faced: the choice of the size of samples inside each stratum and the number of strata is still difficult since the knowledge of data is unknown. To overcome these limitations, we propose a online tree-based data learning strategy to divide data items and allocate appropriate weight for each stratum. In [6], Yan *et al.* proposed an error-bounded stratified sampling technique to minimize the sample size. They need to know the knowledge of data distribution and make sorting for data, which may not be practical for online arriving data. Thus, to make improvement, we design a hash mapping method to stratify the arrived data to the corresponding strata.

Besides, most of studies tend to provide a theoretical error guarantee proved by the probability theory [9], [15]. However, the output quality may not be ensured and it's possible that estimated results are unsatisfactory for customers owing to the probability of sampling. There exist a few quality management strategies to control quality when using approximate techniques from the point of the system level [16]. In our paper, we target the real-time stream processing and also propose a customized error control mechanism to assure output quality.

VII. Conclusion

In this poster, we consider the problems of online data cognition and error control in real-time stream processing. We design an adaptive approximate processing framework to tackle these problems. The framework provides an online learning strategy to relieve the limitation of unknown knowledge for constantly arriving data. Then a dynamic sampling scheme is designed to make a self-adjusting computation. For different user requirements, we propose a customized error control mechanism to detect approximate results. Experiment results with real-world datasets show that our proposed approximate framework can adapt to real-time stream processing and also make efficient approximation with online quality control.

Acknowledgments

This work is supported by National Natural Science Foundation of China (NSFC) (Grants No.61772228), National Key research and Development Program of China (Grant 2017YFC1502306, 2016YFB0701101), Jilin Scientific and Technological Development Program (20160203008GX, 20170520066JH), Graduate Innovation Fund of Jilin University.

References

[1] "CAIDA. Center for applied internet data analysis," http://www.caida.org/home/, [Online; accessed 26-June-2017].

[2] M. Tang and F. Li, "Distributed online tracking," in *Proceedings of the 2015 ACM SIGMOD*. ACM, 2015, pp. 2047–2061.

[3] D. R. Krishnan, D. L. Quoc, P. Bhatotia *et al.*, "IncApprox: A data analytics system for incremental approximate computing," in *International Conference on WWW*, 2016, pp. 1133–1144.

[4] S. Agarwal, B. Mozafari, and A. Panda, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *ACM European Conference on Computer Systems*, 2013, pp. 29–42.

[5] D. S. Khudia, B. Zamirai, M. Samadi *et al.*, "Rumba: an online quality management system for approximate computing," in *International Symposium on Computer Architecture*, 2016, pp. 554–566.

[6] Y. Yan, L. J. Chen, and Z. Zhang, "Error-bounded sampling for analytics on big sparse data," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1508–1519, 2014.

[7] N. Laptev, K. Zeng, and C. Zaniolo, "Early accurate results for advanced analytics on mapreduce," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1028–1039, 2012.

[8] "RIP. Routing information protocol," https://en.wikipedia.org/wiki/Routing_Information_Protocol, [Online; accessed 22-Feb-2018].

[9] S. Lohr, *Sampling: design and analysis*. Nelson Education, 2009.

[10] M. Samadi *et al.*, "Paraprox:pattern-based approximation for data parallel applications," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 35–50, 2014.

[11] "Wikipedia. wikipedia database," http://en.wikipedia.org/wiki/Wikipedia_database, [Online; accessed 28-Oct-2017].

[12] P. S. Efraimidis, "Weighted random sampling over data streams," in *Algorithms, Probability, Networks, and Games*. Springer, 2015, pp. 183–195.

[13] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Very Large Data Bases*, 2006, pp. 607–618.

[14] D. J. e. a. El Sibai R, Chabchoub Y, "Sampling algorithms in data stream environments," in *IEEE ICDEc*, 2016, pp. 29–36.

[15] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 13–30, 1963.

[16] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys*, vol. 48, no. 4, p. 62, 2016.