IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received October 30, 2018, accepted February 4, 2019, date of publication February 25, 2019, date of current version March 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2899825

Online Adaptive Approximate Stream Processing With Customized Error Control

XIAOHUI WEI[1,2], YUANYUAN LIU[1], XINGWANG WANG[1], SHANG GAO[1], AND LEI CHEN[3]

[1]Key Laboratory of Symbolic Computation and Knowledge Engineering, Ministry of Education, Jilin University, Changchun 130012, China
[2]College of Computer Science and Technology, Jilin University, Changchun 130012, China
[3]Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong

Corresponding author: Xingwang Wang (xww@jlu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61772228, in part by the National Key Research and Development Program of China under Grant 2017YFC1502306 and Grant 2016YFB0701101, in part by the Major Special Research Project of Science and Technology Department of Jilin Province under Grant 20160203008GX, in part by the Graduate Innovation Fund of Jilin University, and in part by the Jilin Scientific and Technological Development Program under Grant 20170520066JH and Grant 20190201024JC.

ABSTRACT In approximate processing on stream data, most works focus on how to approximate online arrival data. However, the efficiency of approximation needs to consider multiple aspects. Generally, customers submit their requests with specific quality requirements (e.g., maximum error). This raises a critical problem that online quality control is required to meet the desired quality of service. Since the continuous arriving data may not be entirely stored and needs to be processed immediately, it brings the difficulty of acquiring knowledge online which significantly affects the quality of results. To address these problems, we present an online adaptive approximate processing framework with a delicate combination of data learning, sampling, and quality control. We first design an online data learning strategy for stream data. With the real-time learning results, we propose a dynamic sampling strategy that switches to different sampling methods based on the change of the load. Finally, we present a double-check error control strategy to monitor and correct large errors. Each operation module is correlated through online learning and feedback. The experiments with both synthetic and real-world datasets show that the proposed approximate framework is not only applicable to different data distributions but also provides a customized error control.

INDEX TERMS Approximate computing, online stream processing, sampling, error control.

I. INTRODUCTION

In online stream data processing, one can transform continuous raw data into valuable information, which is widely applied to various fields including online query analysis [1], network traffic monitor [2], and sensor-based measurement networks [3]. Among them, data continuously arrives and users are concerned about real-time results, such as detecting anomalies within a specific time period when monitoring network traffic.

However, explosive data volume and real-time requirements make it challenging to meet the desired Quality of Service (QoS). 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], [5]. Combined with distributed processing models (e.g., MapReduce [6], Spark [7]), approximate computing is attracting more attention to achieve low latency and efficient resource utilization. For instance, when monitoring network flows online, it is not necessary to compute 100% accurate traffic status. The approximate computing can be used to obtain an estimated result, which is often good enough for the analysis of relative throughput. In addition, the use of approximate computing can significantly improve the overall processing performance (also resource utilization).

The most common approximate technique applied in large-scale computing is sampling, which has been extensively adopted in distributed data analytics [7]. With the data knowledge (e.g., data distribution), sampling-based approaches can efficiently process large datasets by computing a subset of data items. However, existing works focus on the choice of approximate methods. To ensure efficient approximate processing, there are three critical problems that need to be considered comprehensively:

- **Data predictability.** General sampling methods are based on the known or predictable data knowledge, such as data distribution, maximum, minimum. Most existing

The associate editor coordinating the review of this manuscript and approving it for publication was Eunil Park.

works assume the characteristics of arriving data can be obtained from historical logs [8]. The preprocessing operation is used to make preparation for data sampling. However, these strong assumptions may lead to inconsistent prediction that produces ineffective samples for processing online stochastic stream data. Since online stream data continuously arrives without being stored and the data knowledge is unknown in advance. Compared with offline data sets, it is difficult to make effective cognition for real-time stream data. Different from assuming a priori knowledge in existing works, we develop an online data learning strategy to better adapt to the dynamic change of stream data.

- **Customer-specific requirements.** Generally, users submit the streaming query to the system with specific requirements. These requirements refer to either desired result accuracy, query response time or available computing resources. Different users and applications have different requirements [9]. Approximate computing should not only improve processing performance, but also ensure the output meets a prescribed level of quality. To adapt to dynamic stream data, the designed sampling strategy should be configurable to satisfy different user/system requirements.
- **Online error control.** In real-time stream applications where data is not being stored, it is needed to check the output to ensure accuracy so that the unsatisfactory results can be corrected in a timely fashion. Currently, main-stream studies tend to provide an error guarantee. In [8], Yan *et al.* proposed an error-bounded stratified sampling method to approximate big sparse data. The error bound denotes an interval where the real value falls in with a high possibility (known as *confidence*). However, due to the probabilistic nature of sampling, it is possible that approximate computing produces unacceptable errors, which will affect the final output quality and reduce customer satisfaction [9], [10]. To make effective output quality, we need to design online strategies to control approximate results for customized requirements.

TABLE 1. Comparison of different approximate frameworks.

Frameworks	Data	Online data learning	Error bounded	Error control
EARL [10]	Static	×	√	√
BlinkDB [4]	Static	×	√	×
ApproxHadoop [19]	Static	×	√	×
IncApprox [7]	Stream	×	√	×
Ours	Stream	√	√	√

As shown in Table 1, several existing frameworks has been proposed to address these problems. However, EARL [10], BlinkDB [4] and ApproxHadoop [19] all aim to address the stored dataset instead of online stream data. EARL utilizes a bootstrap method to estimate the accuracy, in which the number of resamples may be large so that it will result in more overhead. IncApprox [7] is designed to process online arriving stream data but it also does not consider the problems of online data learning and error control.

Taken together, we propose an **online adaptive** approximate stream processing framework with delicate combination of online data learning, sampling and quality control. In the framework, we design three strategies to tackle these problems: an online data learning strategy, a dynamic sampling strategy, and a customized error control strategy. Online data learning is to acquire the knowledge of data distribution, which is dynamically updated to fit the change of stream data. Then the dynamic sampling strategy performs sampling taking into account the fluctuating stream rates. Without requiring application-specific approximate algorithms, the sampling module can execute a self-adjusting computation. Furthermore, the error control strategy leverages an output-based method to detect output and find errors that need to be corrected.

The design goal is to adaptively approximate dynamic real-time stream data even without prior data knowledge and monitor output to implement online quality control. With these strategies, our proposed approximate framework can effectively process online stream data and adaptively meet different QoS requirements. Each component is correlated through continuous learning and feedback to better adapt to the dynamic change in stream computing. Additionally, we give some improvements to optimize the framework to maintain output quality and reduce the overhead of error control. Our **contributions** of this paper are as follows:

(1) We propose an online data learning strategy with a triggered weight update strategy. The scheme can analyze the constantly arriving data to make a weighted stratification for efficient sampling.
(2) We present a dynamic sampling strategy that is able to automatically switches to different sampling methods based on the varying stream rates.
(3) We design a customized error control strategy, which makes an online monitoring for the output errors. The strategy incorporates output's feedback to detect and timely correct large errors.
(4) We conduct extensive experiments with synthetic and real-world data to evaluate and validate the effectiveness of our proposed approximate processing framework.

The remaining of the paper is organized as follows. Section 2 gives a design overview of the approximate framework. Section 3 details the design of online approximate procedure for stream data including online data learning and dynamic sampling strategies. An online error control strategy is introduced in Section 4. Enhancements for the framework is presented in Section 5. Experimental results are shown in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

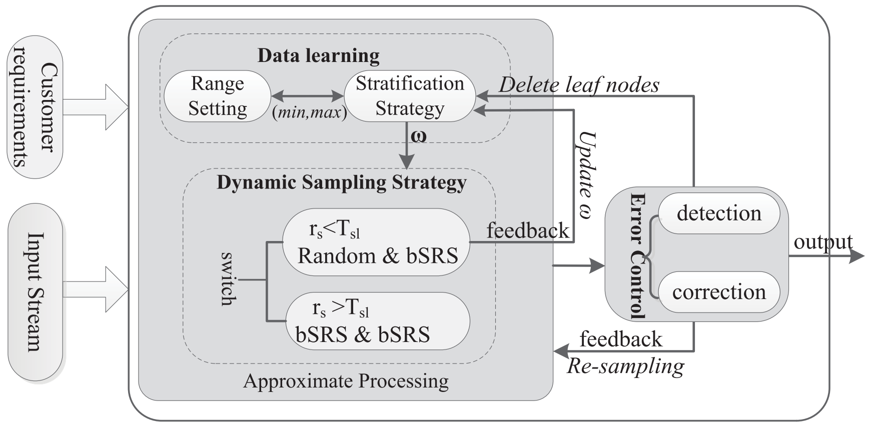


FIGURE 1. A general approximate processing framework over online data streams.

II. OVERVIEW

Figure 1 depicts a high-level block diagram of the approximate framework. Following existing works [7], our computation model for stream processing is based on sliding windows. Given specific user requirements as an input item, the framework makes approximate processing with online data learning and error monitoring. As shown in Figure 1, there are three main components.

A. DATA LEARNING STRATEGY

In this paper, we mainly consider stratified sampling method because it can produce more representative samples with the knowledge of data distribution [8]. To ensure efficient sampling, we first design a stratification strategy for the online learning of knowledge, which can be performed without prior knowledge. Based on the current value range obtained online, the stratification process leverages the binary tree structure to compute the appropriate sampling weights for each stratum. We also present a weight update strategy to dynamically adjust weights, so as to better adapt to the varying characteristics of the stream data. The learning strategy aims to provide preliminary knowledge and make preparation for the subsequent sampling operation.

B. DYNAMIC SAMPLING STRATEGY

We propose a dynamic sampling strategy (DSS), which considers the effect of varying stream rates on the current workload. Based on stratification results, a basic stratified random sampling scheme (bSRS) is designed to get representative samples. Then we dynamically switch to different sampling methods based on the comparison of real-time stream rate r_s and the low rate threshold T_{sl}. There is a switch operation: execute a random sampling and bSRS when $r_s < T_{sl}$; otherwise, execute two bSRS methods. Executing sampling twice is used to compare estimated results in the following error detection module. Moreover, as shown in Figure 1, if $r_s < T_{sl}$, the framework will send a feedback to inform the stratification operation to update weight.

TABLE 2. Frequently used notation.

Notation	Description
L	The height of weight learning tree
(min, mid, max)	Represents the value range of each stratum
ω_{ij}^{k}	the weight of the jth sub-layer of ith node in kth level
T_ω	The threshold to end stratification
m	Number of strata after learning
(a_i, b_i)	Upper and lower bound for ith stratum
r_s	Current stream rate
T_{sl}	Low threshold of stream rate
ε_{S_i}	Estimated error bound for sample S_i
δ_i	Confidence value from sample S_i
$n_w = \lvert window \rvert$	The number of windows with quality checking
t_s	Approximation execution time in a window
t_e	Exact execution time in a window

C. CUSTOMIZED ERROR CONTROL STRATEGY

With the result generated by DSS, we design an online error control component to assess the relative quality of approximate output. Here error control includes two parts: error detection and correction. By theoretically analyzing the probability of a large error, the detection module compares the output results of two sampling methods. If large errors are detected, the correction module utilizes the re-sampling method to correct the output. Moreover, the outputs are detected in a double-check mode. If the output of the current window is first detected, the correction decision is to re-sample data. Otherwise, we also need to adjust weights before re-sampling. The error control module continuously monitors output and provides a feedback to the data learning or sampling module as shown in Figure 1. Due to the difficulty of acquiring exact values for stream data, it is efficient to evaluate output quality by comparing two sampling results.

III. DESIGN OF APPROXIMATE MODULE

In this section, we present the online approximate processing module including online data learning and dynamic sampling strategies. We use the stratified sampling method to approximate stream data. Here, the learning weight information refers to the sampling weight of each stratum [11].

A. ONLINE DATA LEARNING

Firstly, we design a stratification strategy to learn data knowledge for online arrival data. It can be seen as a preprocessing operation. 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 gradually divides the stream according to the value range of data. Then at the end of the partition, each sub-range can be associated with a weight, which will be merged as the final learning result.

1) DATA RANGE UPDATE

Before stratification, the value range of stream data (min, max) need be set. To get the value range online, we utilize the idea of the invalid timer to set the value range. In network routing protocol, the invalid timer specifies how long a routing entry can be in the routing table without being updated [12]. For real data stream, the varying stream means that the value range may change over time. Therefore, to adapt for 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 and the maximum value is set in the same way. There are two parameters: the learning minimum min and the observed minimum min_o that is updated based on the arriving data. Initially, a timer is empirically set and the value range can be set based on historical logs. 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, reset and update min with min_o, the recent minimum value. Through the control of the timer, the value range can be obtained online 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 computation. At first we set the average estimated by random sampling as the reference value, $\hat{v}_{ref}$, which is the comparison sample in the following stratification phase. For random sampling, the sampling weights from the minimum to maximum at this phase are the same.

We denote the tree height of stratification as L. In the first level, data items are divided into two strata according to their value ranges, (min, mid) and (mid, max). With the value of $\hat{v}_{ref}$, we first analyze the weights of these two strata, denoted as ω^1_{11} and ω^1_{12}. After stratification, the average value of each stratum can be obtained based on data items from each stratum, denoted as $\hat{v}^1_{11}$ and $\hat{v}^1_{12}$. According to the

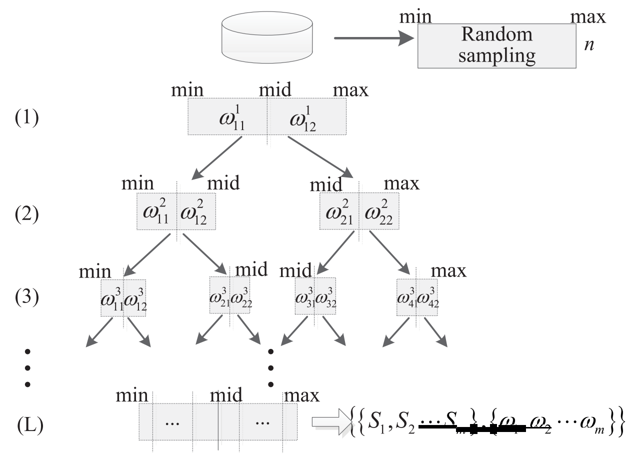

min
max
Random sampling
n
(1)
min
mid
max
ω^1_{11} ω^1_{12}
(2)
min
mid
mid
max
ω^2_{11} ω^2_{12}
ω^2_{21} ω^2_{22}
(3)
min
mid
mid
max
$\omega^3_{11}\omega^3_{12}$
$\omega^3_{21}\omega^3_{22}$
$\omega^3_{31}\omega^3_{32}$
$\omega^3_{41}\omega^3_{42}$
(L)
min
mid
max
...
...
$\{\{S_1, S_2 \cdots S_m\}, \{\omega_1, \omega_2 \cdots \omega_m\}\}$

FIGURE 2. The process of stratification with the binary tree.

sampling theory [11], the definition of 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 and $\sum \pi_i = 1$. If two strata have the same inclusion probability ($\pi_1 = \pi_2 = \frac{1}{2}$), the overall average value is

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

where $\hat{v}_{ref}$ is computed as a reference value to adjust the weights of strata. Compared with the initial value $\hat{v}_{ref}$, the weight of each sub-range (ω^1_{11} or ω^1_{12}) can be updated. Denote β as the proportion tuning parameter and we use β to adjust the values of ω^1_{11} and ω^1_{12}. Through comparing $\hat{v}^1_{11}$ and $\hat{v}^2_{11}$, there are:

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

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

Conditions (1) and (2) correspond to different proportions of two sub-ranges. Hence, the weight values at these two strata can be modified based on conditions (1) and (2). As shown in Figure 2, each stratified value range is further divided into two child nodes with smaller value ranges in the next level. We use the first level as an example to introduce the stratification method. Then we use Algorithm 1 to summarize the stratification strategy. For each child node, a smaller stratum will get a new weight value using the same method as described in the first level. The modified weights at the first level will be set as the reference weights of the next stratified sampling until the partition ends. Following the example, in the second level, v^1_{11} is used to compare the average estimated by random sampling. Then we estimate ω^2_{11} and ω^2_{12} based on the estimated value of each smaller stratum $\hat{v}^2_{11}$, $\hat{v}^2_{12}$. The computation of the tree height L depends on the difference between the maximum and minimum value of the sub-range. During the weight computing phase, a weight learning threshold T_ω is needed to judge whether to end the

Algorithm 1 Stratification strategy

```
1:  for $k \leq L-1$ do
2:      Divide the value range of each node (level $k$) into two child nodes with smaller value ranges in the next level (level $k+1$);
3:      set $\hat{v}_{ref}$ as the average estimated by random sampling;
4:      for $i = 1$ to $2^k$ do
5:          if $\hat{v}_{ref} \geq \hat{v}^k_{i1}$ then // If dividing the right child, we compare $\hat{v}^i_{i2}$ with $\hat{v}_{ref}$.
6:              compute $\beta$ as $(\frac{1}{2}+\beta)\hat{v}^{k+1}_{i1} + (\frac{1}{2}-\beta)\hat{v}^{k+1}_{i2} = \hat{v}_{ref}$;
7:              $\omega^1_{i1} = \frac{1}{\frac{1}{2}+\beta}, \omega^1_{i2} = \frac{1}{\frac{1}{2}-\beta}$;
8:          else
9:              compute $\beta$ as $(\frac{1}{2}-\beta)\hat{v}^{k+1}_{i1} + (\frac{1}{2}+\beta)\hat{v}^{k+1}_{i2} = \hat{v}_{ref}$;
10:             $\omega^1_{k1} = \frac{1}{\frac{1}{2}-\beta}, \omega^1_{k2} = \frac{1}{\frac{1}{2}+\beta}$
11:         end if
12:     end for
13: end for
```

Algorithm 2 A Basic Stratified Reservoir Sampling Algorithm (**bSRS**)

```
Input: a real-time data stream, sample size $n$, weight set $\{\omega_i\}$
1:  sample ← ∅;
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:              Add the item to sample[$i$];
10:         else
11:             $p \leftarrow \frac{n_i}{|S_i|}$;
12:             Replace a random item from sample[i] with a probability $p$;
13:         end if
14:     end for
15: end for
```

process of weight update. T_ω can be obtained by evaluating the difference of weights before and after the update. Eventually, at each phase, the weights are modified as stream data constantly arriving.

In the end of stratification, our proposed divide-and-conquer method partitions the value range of 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 with the updated range to generate more representative samples for the recently arriving data.

B. DYNAMIC SAMPLING STRATEGY

The approximate processing module shown in Figure 1 describes a complete execution for online data streams. The current stream rate is used to trigger the switch of approximate methods. The designed strategy considers the change of the load which depends on fluctuating stream rates. 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. In the approximate module, we design a dynamic sampling strategy (DSS) that switches to different sampling methods to avoid adding extra overhead during the peak of data arrival.

We first introduce a basic stratified sampling algorithm that will be used in DSS. Algorithm 2 describes a basic stratified reservoir sampling algorithm (bSRS). In the algorithm, we leverage a hash mapping method to stratify the arriving stream. Based on the weights obtained from the stratification strategy, it allocates the sample sizes for each stratum and then fills each sample set using a conventional reservoir sampling (CRS) [15]. 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. If accepted, we replace a randomly selected item from the sub-sample set with the arriving item. To compute the sample size as an input, we can adopt existing resource prediction models [13], [14]. They model a virtual function that converts the user-specified query budget (computing resources, response time) to the number of items to be processed (sample size).

The proposed algorithm bSRS can get a representative sample of stream data and the dynamic switching can quickly accommodate the fluctuation of data streams. It can avoid extra overload at the peak of data streams since using random sampling consumes more resource than stratified sampling. To introduce DSS, we first use an example to illuminate the setting of T_{sl}. Assume the scale of system process capacity is 2000. If the estimation result shows when the real-time data volume is large than 500, executing random sampling will increase the overhead of data processing. Then we can set the switching threshold of dynamic sampling strategy as 500 that can be converted to the rate threshold T_{sl}. The dynamic sampling strategy (DSS) is described as follows:

(1) If $r_s < T_{sl}$, we select to perform both the random sampling and bSRS algorithms concurrently and then give a feedback to the stratification module.
(2) If $r_s \geq T_{sl}$, execute two bSRS sampling algorithms concurrently.
(3) Return the result to the error detection module.

Executing sampling twice aims to provide the result comparison of two sampling methods, which is used to detect the probability of error occurrence in the design of the online error control strategy (Section IV). When the stream rate is low, the switch to different sampling methods is to present feedback for weight adjustment for the stratification phase when the current load is relatively small. If the comparison result has a big deviation, the data learning module will be triggered to update weights (Section V-A).

Next, we analyze the probabilistic nature of sampling according to the probability theory [11]. Assume we compute an AVG value for stream data and denote the approximate value as $\hat{\hat{v}}$, the exact value as $\bar{v}$. According to the Hoeffding inequality [16], the estimated error bound is no more than ε:

$$P(\left|\hat{\hat{v}} - \bar{v}\right| \geq \varepsilon) \leq 2e^{-\frac{2n\varepsilon^2}{(b_i - a_i)^2}} \tag{2}$$

where a_i and b_i are lower and upper bounds of values. We use Eq. (2) to estimate error bound when given sample size n. This probability can be interpreted as the level of significance α (probability of making an error) for a confidence interval around $\hat{\hat{v}}$ of size 2ε, that is:

$$\alpha = P(\bar{v} \notin \left|\hat{\hat{v}} - \varepsilon, \hat{\hat{v}} + \varepsilon\right|) \leq 2e^{-\frac{2n\varepsilon^2}{(b_i - a_i)^2}} \tag{3}$$

where the confidence $\delta = 1 - \alpha$. We analyze the probability to explain the cause of error control in the next section. Then, the estimated error bound is:

$$\varepsilon = (b_i - a_i)\sqrt{\frac{1}{2n} \ln \frac{2}{1-\delta}} \tag{4}$$

If the sample size n is larger than the total size, we will consider the whole data set for sampling. Based on the above discussion, next we make a quantitative analysis for the superiority of the proposed stratified sampling method.

Theorem 1: Given a sample size n and confidence δ, theoretically the error bound ε_S generated from bSRS and ε_0 from random sampling satisfies: $\frac{\varepsilon_S}{\varepsilon_0} \leq \frac{1}{\sqrt{m}}$

Proof: When random sampling, according to Eq. (4), $\varepsilon_0 = (b_i - a_i)\sqrt{\frac{1}{2n} \ln \frac{2}{1-\delta}}$.

When using the improved stratified sampling method, assume S_i contains N_i values that corresponds to the sub-sample size n_i with its error bound ε_i. We have:

$$N = N_1 + \cdots + N_m \tag{5}$$

$$n = n_1 + \cdots + n_m \tag{6}$$

$$\varepsilon_S = \frac{N_1 \cdot \varepsilon_1 + \cdots + N_m \cdot \varepsilon_m}{N} \tag{7}$$

Next, we extend Eq. (7) using the Hoeffding inequality and weight values:

$$\begin{aligned}\varepsilon_S &= \frac{(b_i - a_i)}{mN}\left[N_1\sqrt{\frac{\omega_1}{2n} \ln \frac{2}{1-\delta}} + \cdots + N_m\sqrt{\frac{\omega_m}{2n} \ln \frac{2}{1-\delta}}\right] \\ &= \frac{(b_i - a_i)}{m}\sqrt{\frac{1}{2n} \ln \frac{2}{1-\delta}}\left(\sqrt{\frac{1}{\omega_1}} + \cdots + \sqrt{\frac{1}{\omega_m}}\right)\end{aligned} \tag{8}$$

The sampling theory defines the definition of the sampling weight is the reciprocal of the inclusion probability: $\omega_i = \frac{1}{\pi_i}$, and $\pi_i = \frac{n_i}{n}$. It can be seen that the proportion of N_i to N is nearly consistent with that of n_i to n. Therefore, we combine Eq. (4) and Eq. (7) to model the relation of ω_i and n. Then when comparing ε_S with ε_0:

$$\frac{\varepsilon_S}{\varepsilon_0} = \frac{1}{m}\left(\sqrt{\frac{1}{\omega_1}} + \cdots + \sqrt{\frac{1}{\omega_m}}\right) \tag{9}$$

Since there is the equation $\frac{1}{\omega_1} + \cdots + \frac{1}{\omega_m} = 1$, with the principle of inequality we have:

$$\sqrt{\frac{1}{\omega_1}} + \cdots + \sqrt{\frac{1}{\omega_m}} \leq \sqrt{m} \tag{10}$$

Combining the Eq. (9) and Eq. (10) we finish the proof. □

Theorem 1 makes quantitative analysis that sampling with the random method need more data than bSRS when given an error bound. In DSS, We use the low threshold rate r_s as a criteria to judge whether to switch the sampling method. If the current stream rate $r_s < T_{sl}$, we perform both the random and bSRS algorithms. Hence, the setting low threshold rate affects the total sample size. The smaller the low threshold rate is, the smaller the total sample size needed is.

IV. CUSTOMIZED ERROR CONTROL STRATEGY

Although these approximate techniques can provide significant performance gains, it is still difficult and expensive to monitor the output quality, especially for online stream data. Existing work [10] shows that an exact bootstrap for accuracy estimation needs $\binom{2n-1}{n-1}$ resamples where n is the sample size. Moreover, different applications also have different demands for output errors: some requests are submitted with the constraint of maximum error while some may concern with the average error.

Currently, most research 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. Generally, the error guarantee given by the error estimation theory showed in Equation (4) represents the overall average error value. Owing to the probability of sampling, it may be that the output error exceeds the specified error bound ($\bar{v} \notin \left|\hat{\hat{v}} - \varepsilon, \hat{\hat{v}} + \varepsilon\right|$), which has been explained in Eqs. (2) and (3). In this case, the accuracy requirement of output results may still be unsatisfactory from customers' view. Thus, in order to not affect the final output quality, we need to reduce these unacceptable results. In particular, it is also necessary to online check the output for real-time stream processing.

With different user requirements, we present a customized error monitoring strategy to detect and correct approximate outputs. First, we design a double-check error detection strategy is designed to assess the relative quality of the approximated output results. The main idea is to analyze the probability of occurrence of large errors through detecting output results.

Error detection To evaluate and manage the output quality, it is important to ensure the error detection model has low overhead. In our design, we leverage multiple samples generated in the sampling stage, to evaluate the output quality. Relative to comparing with the total dataset, it is light-weight to make comparison within different samples. In the following, we consider the comparison between two samples.

Assume the above approximate module simultaneously generates two samples, denoted as S_1 and S_2, with two estimate values, $\hat{\bar{v}}_1$ and $\hat{\bar{v}}_2$. According to the Eq. (3), with the confidence δ the estimation's error $\left|\hat{\bar{v}}_1 - \bar{v}\right|$ or $\left|\hat{\bar{v}}_2 - \bar{v}\right|$ satisfies:

$$\left|\hat{\bar{v}}_1 - \bar{v}\right| \leq \varepsilon_{S_1}, \quad \left|\hat{\bar{v}}_2 - \bar{v}\right| \leq \varepsilon_{S_2} \tag{11}$$

where $\bar{v}$ is the exact value and ε_{S_1}, ε_{S_2} are the corresponding error bounds for samples S_1, S_2, respectively. Then we use Eq. (11) to concatenate two estimate results. To eliminate the limitation of the unknown of exact results, we consider the sum of two estimation's errors:

$$\begin{aligned}&\left|\hat{\bar{v}}_1 - \bar{v}\right| + \left|\hat{\bar{v}}_2 - \bar{v}\right| \leq \varepsilon_{S_1} + \varepsilon_{S_2}\\ &\quad \Longrightarrow \left|\hat{\bar{v}}_1 - \bar{v}\right| + \left|\bar{v} - \hat{\bar{v}}_2\right| = \left|\hat{\bar{v}}_1 - \hat{\bar{v}}_2\right| \leq \varepsilon_{S_1} + \varepsilon_{S_2}\end{aligned} \tag{12}$$

For simplicity, let $\Delta = \varepsilon_{S_1} + \varepsilon_{S_2}$ represent the error deviation that can be 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. (12), then the detection module will report an error. For the average error, the module will compute the average of multiple comparison results [4].

We assume the sampling phase produces the same error-bound samples, which corresponds to the equation $\varepsilon_{S_1} = \varepsilon_{S_2} = \varepsilon_S$.

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 the data of the current processing window, the output quality can be improved by timely correcting unsatisfactory results.

A. ERROR CONTROL STRATEGY

Next, we propose a scheme for the output-based error detection and correction as shown in Figure 3. Our proposed approximate schema first generates two samples for comparison, and then output the computed values with their estimated error bounds. The computed results and corresponding error

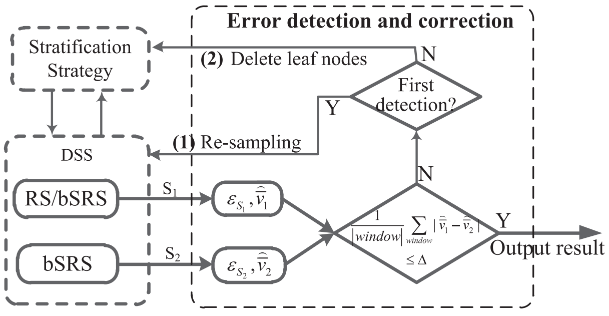

Error detection and correction
Stratification Strategy
(2) Delete leaf nodes
N
First detection?
Y
(1) Re-sampling
DSS
N
RS/bSRS
S1
$\varepsilon_{S_1}, \hat{\bar{v}}_1$
bSRS
S2
$\varepsilon_{S_2}, \hat{\bar{v}}_2$
$\frac{1}{|window|}\sum_{window}|\hat{\bar{v}}_1 - \hat{\bar{v}}_2| \leq \Delta$
Y
Output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

FIGURE 3. The error detection and correction strategy.

bounds are the inputs of error detection module. For the maximum error, we need to detect the output result of each window. If the difference between 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 within the given time period. As shown in Figure 3, $|window|$ denotes the number of detection windows. 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 process aims to provide quality-assurance output. When monitoring a large error, we will judge whether it is detected for the first time. If so, the decision is to re-sample the data. Otherwise, the module will feedback to the stratification strategy for weight adjustment. When the error is detected for the second time, it means the large error cannot be well corrected only through re-sampling, and the weights of sampling also need to be improved to generate more accurate results. The process is described in Figure 3 that is the refinement of the error control module shown in the right of Figure 1. Before re-sampling, we delete the leaf nodes in the stratification binary tree and go up to select their parent nodes as the reference of sampling weights. Here the weights obtained by leaf nodes are considered inappropriate. 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. Moreover, the error detection strategy cannot only monitor output quality for error detection and correction, but also ensure low extra overhead. As previously mentioned, executing two sampling schemes concurrently aims to update weights and compute the value of Δ. The results of error detection are beneficial to update weights for better approximate outputs in turn. 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.

In the following, we theoretically analyze the improvement of output quality when using the error correction strategy. Denote the confidence values of the two sample sets as δ_1, δ_2. The error detection module can directly output if the actual approximated results both have an error value within the estimated error bound, which is formulated by Eq. (12). Otherwise, the error detection module switches to re-sampling process when the results cannot satisfy Eq. (12). In theory, based on the given confidence values, when assuming that $\delta_1 = \delta_2 = \delta$, the improvement of output quality can be quantified as below.

Theorem 2: With the error detection and correction strategys, 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 $1-(1-\delta)^2-\delta = \delta(1-\delta)$. □

For instance, assume the specified confidence $\delta = 90\%$. With our proposed quality monitoring strategy, theoretically the probability within the given error bound will be raised from 90% to 99%.

In practice, there are two situations that the error detection model may fail to find large errors. In this case, the output results may also be unsatisfactory with large errors. To better explanation, these situations can be expressed as:

(1) $\left|\hat{\bar{v}}_1 - \hat{\bar{v}}_2\right| \leq \Delta$ but $\hat{\bar{v}}_1 - \bar{v} > \varepsilon_{S_1}, \hat{\bar{v}}_2 - \bar{v} > \varepsilon_{S_2}$

(2) $\left|\hat{\bar{v}}_1 - \hat{\bar{v}}_2\right| \leq \Delta$ but $\bar{v} - \hat{\bar{v}}_1 > \varepsilon_{S_1}, \bar{v} - \hat{\bar{v}}_2 > \varepsilon_{S_2}$

Under these conditions, both two compared samples output undesired results and the proposed model cannot detect the large errors. However, the given approximate method provides an error bound guarantee and the probability of the occurrence of the above cases is low. Theoretically, it is much less than $(1-\delta)^2$.

V. ENHANCEMENTS AND ANALYSIS

To better adapt to the varying stream data, we additionally present some improvements to dynamically maintain the effectiveness of the approximate framework and further optimize the cost of approximate processing.

A. TRIGGERED MAINTENANCE OF STRATIFICATION WEIGHT

In DSPS, we set the whole process of stratification above as a learning part of approximate computing framework. Owing to the continuous arriving data, the knowledge obtained from input data is gradually accumulated and the result of data learning should be updated. Hence, we need to dynamically adjust initial weights based on the feedback of sampling results, so as to better adapt to the characteristics of stream data and assure the accuracy of approximate results.

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

(a) As described in Section III-B, when the data arrival rate is below the specified rate threshold T_{sl}, it can be triggered to adjust weights of each stratum. Since the low arrival rate means relatively small computation overhead, the sampling weights can be maintained at this stage.

(b) When the output detection module finds that approximated results have large errors, it can be triggered to adjust weights to improve the accuracy of sampling results.

Figure 1 shows two situations triggered to inform the data learning module for weight update. With the above-listed trigger conditions, the trigged weight update (TWU) process is described in Algorithm 3. To ensure low update overhead as possible, we do not need to re-calculate all weights. In section III-A, our proposed stratification strategy is based on the binary tree structure. During the stratification tree, the parent node produces two child nodes with two sub-strata. When updating the weights of the child node, we can back to its parent node (line 2) and recalculate weights of sub-strata based on current data items (line 4). Therefore, the weight can be updated by their parent (line 7) or ancestor nodes of the upper levels, which is a layer-by-layer operation. Besides, the modification on L_c in TWU at line 7 can also be set as $L_c = L_c - 2$ to faster end the loop operation of line 5.

Algorithm 3 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 the results with the random and stratified sam-
    pling 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 described in Section III-A.
11: Return the updated weights to the DSS module.
```

B. IMPROVEMENT WITH STREAM EVOLUTION

As new stream data arrives, the learning result of stratification tends to be stable and the corresponding output results will satisfy customer requirement with higher probabilities. At this moment, some improvements can be designed to reduce the overhead of online approximate processing.

First, we can make improvement by decreasing the frequency of weight update. In our design, the stratification module will receive a feedback when the stream rate is lower than T_{sl}. With the stream evolution, the feedback can be reduced to lower the frequency of weight update. In this way, the weight update feedback is not always triggered when $r_s < T_{sl}$. We can set a count control parameter at the DSS and selectively provide a feedback to the stratification module.

Second, the frequency of error control can also be reduced as data continuously arriving. The process of comparing two estimated results can be executed in a sampling way. In DSPS, our computation model is based on the sliding window. Videlicet, it is not necessary to constantly monitor the quality and we can compare results to detect errors generated by partial windows. Referring to the idea in Paraprox [17], it can be implemented by setting a fixed checking interval N and checks are performed every N^{th} invocation. At other time, we do not need to execute sampling twice.

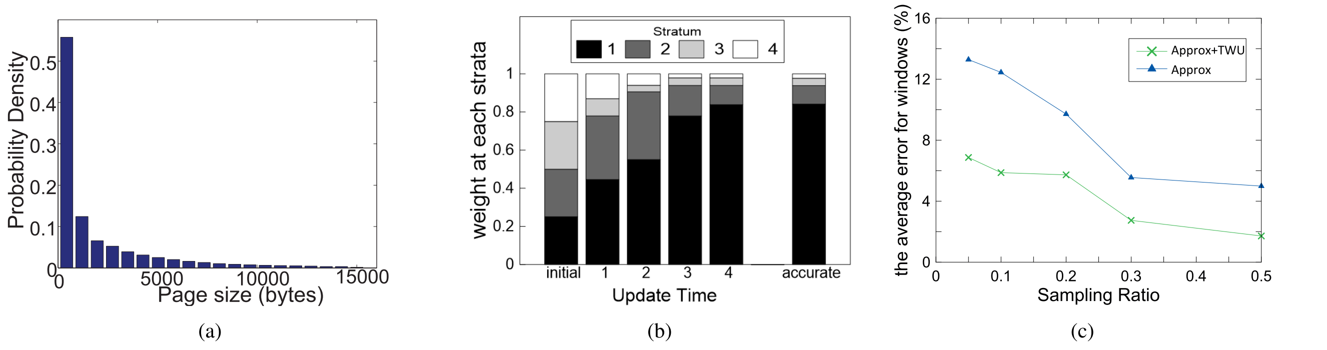


FIGURE 4. (a) The probability density of datasets. (b) Weight learning result (c) Comparison with and without triggered weight update with stream rate threshold.

These improvements are designed to further reduce the overhead of online learning and quality control. Compared with the difficulty of correcting the approximate outputs after processing, it is light-weight to execute sampling twice for quality checking.

C. ANALYSIS OF COMPUTATIONAL COST

To validate the efficiency of the framework, next we analyze the overhead of the approximate strategy through computing the execution time. Eq.(13) shows the ratio of the total execution time between approximate and exact computation. t_e is the execution time of one exact processing, and t_d is the error detection time. The numerator indicates the approximation overhead including extra cost generated by the error control strategy. n_w is the number of windows that executes sampling twice.

$$Cost_{approx} = \frac{(N + n_w) \times t_s + t_d}{N \times t_e}, \quad t_s = \frac{t_e}{G(ratio)} \tag{13}$$

In parallel distributed execution environment, it is commonly observed that processing time grows linearly with input size for the majority of I/O bounded queries [4], [18]. Based on this, we can estimate t_s using the time of exact execution (t_e) and approximation gains ($G(ratio)$). We use $G(ratio)$ to represent the inversely linear proportion between the sampling ratio and approximation gains. Compared with the actual execution, the time of comparing sampling results to detect error t_d is negligible.

VI. PERFORMANCE EVALUATION

We implement the proposed online approximate strategies and evaluate their performance both with the synthetic and real-world datasets. To analyze the effectiveness of the online approximate and error detection strategies, we drive experiments with the online aggregation operation, AVG and the other common aggregation operators (SUM, COUNT, RATIO) are similar. We used two datasets:

(1) *Synthetic data*: To evaluate the results of different data distributions, four classical probability distributions are tested including beta distribution, normal distribution, uniform distribution and Zipf distribution. We generate 1,000,000 data items for each kind of distribution to show the effects of online data learning.

(2) *Real-world data*: We study an available large-scale data analysis application, WikiLength [20], which analyzes lengths of web pages (bytes) [21]. The real dataset includes the December 2016 snapshot of Wikipedia, which contains millions of English articles in XML format. The data file is a bzip2-compressed type with 12.6GB.

We respectively examine the effects of varying different parameters in our evaluation, including the sample ratio, window size, weight learning threshold and maximum/average error deviations.

In the settings, we use the *Sine* function to simulate the arrival rate of stream data, which refers to real-time analysis applications mentioned in [22]. Data is processed in the corresponding sliding window, and we analyze the average page length in each processing window so that the real-time query results can be obtained. For simplify, the statistical error of the entire dataset denoted as ε_T and the current sliding window denoted as ε_w are both considered for real-time approximate processing.

A. STRATIFICATION LEARNING ANALYSIS

In this section, we mainly test the performance of the proposed approximate framework through real datasets. The AVG operation selectively computes web pages whose values range from 1 to 15,000 bytes and we fix the confidence to 95%. We first construct a binary tree for online data learning. In each partition, the weight update is computed as data continuously arriving, and the difference before and after the update is compared to judge whether to end each partition. In practical applications, it is also advisable to end partition when the weight of the corresponding value range changes tinily.

The accurate probability density distribution in Figure 4(a) indicates that data is highly skewed. Figure 4(b) 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. A three-level binary tree is constructed and the value range of page length is

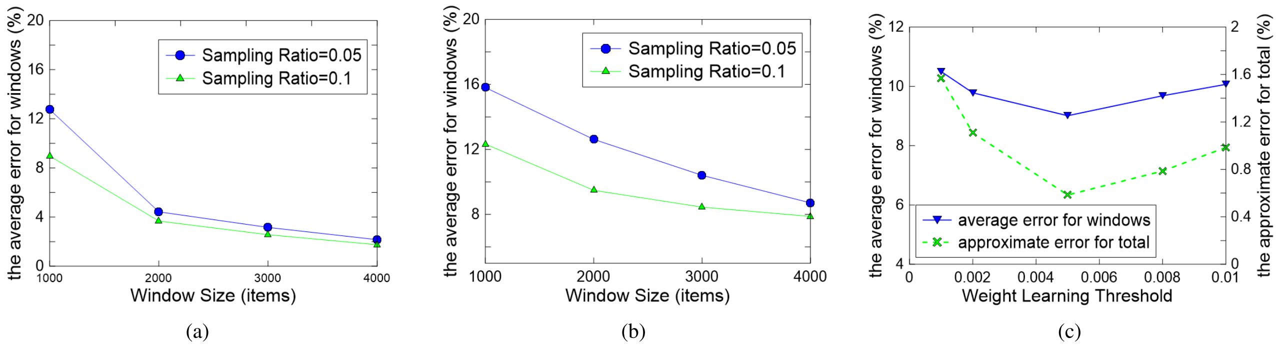


FIGURE 5. **(a)&(b) Effects of window size. (c) Weight learning result.**

partitioned into four strata. The accurate ratio of each stratum is listed in the rightmost histogram of Figure 4(b). At the end of the learning phase, the estimated weight in the first stratum is about 0.8382 compared with the accurate value 0.8405. We can see that the weight learning process gradually closes to the exact values and the final result is nearly consistent.

In experiments, the weight update will combine the weight information of the previous state with the current data distribution where we set a parameter to trade off these two factors. When the data volume increases, it is possible the initial learning result may have some deviation, but the deviation can be updated by the later triggered update. Figure 4(c) shows the comparison result before and after adding triggered weight update with stream rate threshold T_{sl}. It can be seen that the triggered update operation can assure more accurate approximate results.

B. EFFECTS OF WINDOW SIZE

Considering the stream processing model, we analyze the effects of varying window sizes on sampling. Through setting different window sizes, we measure the average approximate error under online processing windows. Here the window size increases from 1000 to 4000 items. To better validate the impact of window sizes, we respectively test the statistical errors for windows ε_w with different sampling ratios and data volumes. Each time the experiment is repeated 20 executions and then we compute an average value.

As shown in Figure 5, we compare approximate errors under different sampling ratios (0.05 and 0.1). At the small data scale shown in Figure 5(a), when setting the sampling ratio to 0.05, the approximate accuracy computed under each processing window increases by about 83%. For instance, when the window size becomes 4000, the approximate error reduces to 0.02. The larger data scale in Figure 5(b) can also demonstrate the effect. The reason is that the larger window sizes can obtain a more comprehensive data information, which is critical for a good learning result, and the sampling result directly depends on stratified weight values. However, it is bounded and Figure 5(a) & 5(b) illustrates that the trend of improvement is gradually slow.

When considering different data volumes, the value of ε_w with a small data volume shown in Figure 5(a) is lower than that in Figure 5(b). According to the compared results, if setting window size to 4000 and sampling ratio 0.1, we can see that the average error ε_w with smaller volume is about 2.16% but with large volume reaches 8.44%. For the same window size, the proportion of data with small volume is larger than that with large volume. The stratification strategy will acquire relatively comprehensive knowledge in a larger window to obtain more accurate outputs.

C. EFFECTS OF WEIGHT LEARNING THRESHOLD

During the stratification phase, we utilize the arriving data to adjust weights from the upper to lower nodes. To judge whether to end the adjustment, an appropriate weight learning threshold T_w needs to be set. In this section, we vary the value of T_w from 0.001 to 0.01 to choose the best threshold by evaluating ε_w and ε_T.

Figure 5(c) lists the average error for windows (Y-axis on left) and the total approximate error (Y-axis on right). When T_w increases from 0.001 to 0.01, the values of statistical error have the same changing trend, which both decrease and then increase. Especially, the overall error will reduce by 68% (from 1.56% to 0.58%) when T_w varies from 0.001 to 0.005. If T_w is set too small, the weight of each stratum may be adjusted higher than the accurate result since smaller thresholds cause overfitting for those strata with heavy weights. Then, the statistical errors trend to rise when T_w increasing from 0.005 to 0.1. The reason is that setting lager thresholds will cause insufficient learning process where weights are estimated with more deviations. Therefore, it is significant to choose appropriate learning thresholds for more efficient approximate processing.

D. EFFECTS OF DIFFERENT DISTRIBUTIONS

To validate the effectiveness of the online approximate strategy, we generate four datasets that respectively correspond to different common distributions as shown in Figure 6(a). As a comparison, we measure the average errors both with random and stratified sampling (bSRS). The sampling ratio varies from 0 to 0.5.

From Figure 6(b) and 6(c), we can see that the sampling results are different with different distributions. The results demonstrate that stratified sampling performs better than

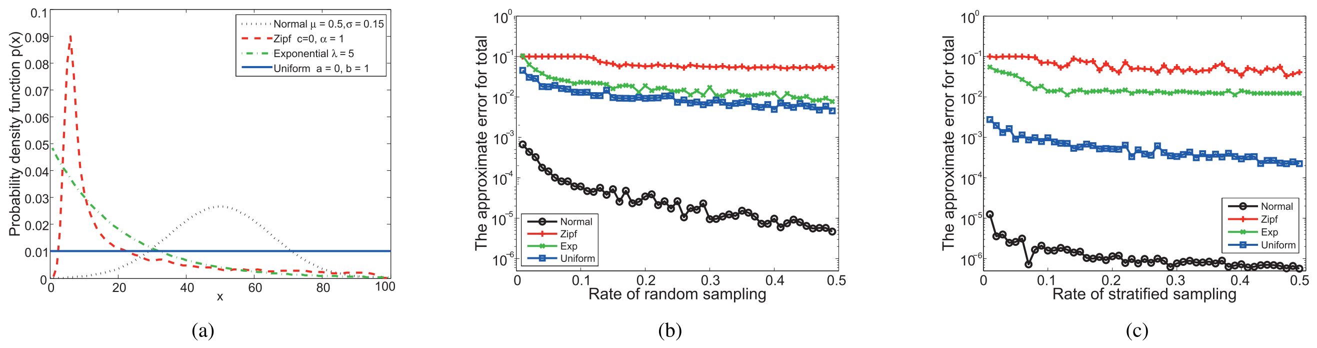


FIGURE 6. **(a) Different common distributions. (b)&(c) Error comparison with random and stratified sampling.**

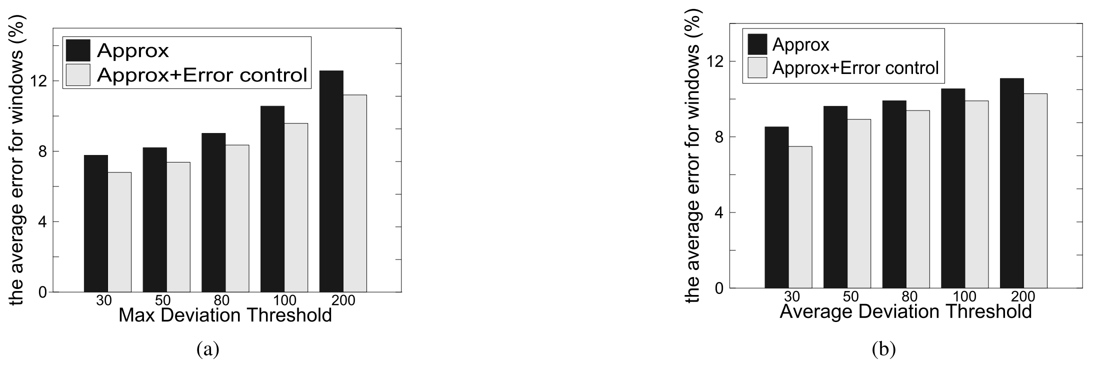


FIGURE 7. **(a)&(b) Effects of different max deviation thresholds Δ.**

random sampling for these distributions, which also reflects the effectiveness of our proposed approximate processing strategy. However, for the exponential distribution, when the sampling ratio is larger, random sampling performs better because it can uniformly sample the data from the head and tail.

E. EFFECTS OF ERROR CONTROL

In this section, we evaluate the performance of the error control strategy. To meet the customer requirements, different deviation thresholds Δ are set. We test the variation of ε_w with different thresholds Δ, which reflects the effect of quality control in the case of the unknown of exact results. Then we compare approximate results with and without the error control to show the effect of online error control.

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 as 1% and Δ varies from 30 to 200. First, we set Δ as the maximum deviation threshold and compare whether the difference between two sampling results generated exceeds Δ. As shown in Figure 7(a), our error control strategy can reduce the average window error since it can detect large errors and make timely correction by sampling. Then we consider Δ as the average deviation threshold to test the effect of average error constraints. Figure 7(b) depicts the error for windows when each time detecting the average error of the recent five processing windows. Similar to setting a maximum threshold in Figure 7(a), the error control strategy can reduce overall errors. Experiments indicate that online error detection can decrease accuracy loss and improve output quality.

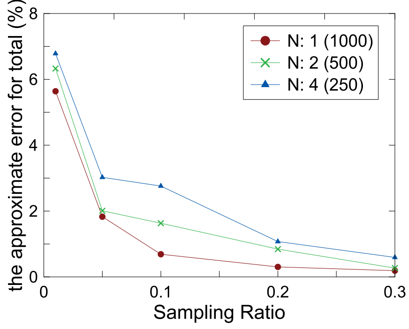


FIGURE 8. **Effects of varying the error control frequency N^{th}.**

As mentioned in section V-B, the overhead of error control can be optimized through performing error checking every N^{th} invocation. Next, we analyze the effects of setting different N (e.g., 1, 2, 4) to ε_T when varying the sampling ratio. As shown in Figure 8, 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 extra increased processing window used to detect error is 1000, and so on.

To verify the comparison listed in Table 1, we also assess the effect of error control compared with IncApprox [7]. With the same parameters, Figure 9 shows the comparison of accuracy loss with different sampling ratios when sampling with the methods of IncApprox and the addition of the error control strategy. The result also indicates that the error control strategy can contribute to a better approximate result.

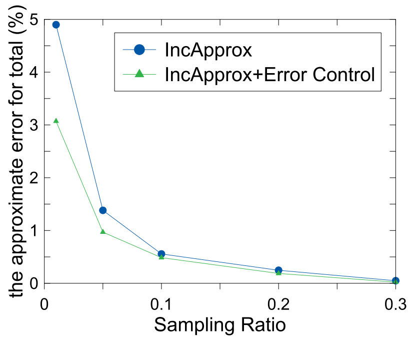


FIGURE 9. Error comparison with IncApprox and the addition of error control.

In actual stream applications, it is impractical to know accurate results when using approxiamtion. Therefore, our proposed error control scheme cannot detect all situations when large errors occur. In the same way, some detected large errors may not be actually a large error. Fortunately, the probability of the occurrence of this case is low. The extra cost caused by unnecessary re-sampling in this case can be ignored.

F. COMPARISON WITH RANDOM SAMPLING

Next, we evaluate our proposed sampling algorithm (bSRS) by comparing with random sampling. In this group of experiments, the value of T_w is set to 0.005. We test the average error of windows ε_w that reflects the performance of real-time processing and also record the approximate error for the total input stream data ε_T. Setting different window sizes can also affect the results of sampling shown in section 6.2. Thus, we set two window sizes (1000 and 2000) to separately generate approximate results.

Figure 10 depicts the comparison results of two sampling methods with different window sizes. When increasing the sampling ratio from 0.05 to 0.5, the value of ε_w reduces with two sampling schemes. ε_w decreases from 0.044 to 0.001 when the sampling ratio varies from 0.05 to 0.5. With our stratified method, the error is lower than that with the random method. Based on the results of learning weights, the stratified scheme can obtain a more representative sample. Above all, when the window size is larger, the sampling error will further decrease shown in Figure 10(a). For example, if setting the sampling ratio to 0.2, the value of ε_w is about 0.064 with window size 1000 while it reduces to 0.023 in a larger size. The statistical error for the total ε_T is compared in Figure 10(c). Since ε_T is analyzed equivalent to sample in the whole static dataset, ε_T is relatively lower compared with ε_w, which is an online statistical operation.

G. EVALUATION OF COMPUTATIONAL COST

We compute the computational cost to evaluate the relative gains of approximation. Here we set the sampling ratio 1% and vary N from 1 to 20. Figure 11 illustrates the computational cost with different quality checking intervals. We can see that the approximation cost decreases as increasing the checking interval, which further reduces the extra cost brought by evaluating the output quality. For instance, when comparing results of a window in an interval ($|window| = 1$), the approximation cost decreases from 1.87% to 0.97%. The result shows detecting more windows in an interval also increases the computation cost.

In conclusion, the above experiment results show the effectiveness of our proposed online approximate stream processing framework. It is applicable for different data distributions. By setting different learning parameters, the combination of online data learning and stratified sampling can make an efficient approximation. Compared with other frameworks, it also provides a customized error control to correct large errors online.

VII. RELATED WORK

There have been numerous researches about approximate computing for big data processing. Approximation techniques such as sampling [23], [24], sketches [25], and online aggregation [26], have been extensively studied to achieve high-efficiency processing. Among them, the sampling technique is most often applied to generate data summaries and make approximate computation. In this section, we discuss the sampling technique used for large-scale stream processing.

Many sampling methods are first proposed to process static data, like those stored in database systems [27]. Conventional methods include random sampling, weight sampling [28], stratified sampling [29], etc. To ensure a better approximation results, different improved sampling schemes have been proposed to be more suitable for different situations in research literature [1], [3], [30]. For instance, Roy *et al.* proposed a distributed stratified-sampling method which partitions the surveyed population into homogeneous subgroups over social networks used in the MapReduce framework [31]. With the prior knowledge of stored data sets, appropriate sampling methods can be applied for more efficient queries. Most works create samples based on the assumptions the future workload is predictable through historical usage [4]. With sampling, sometimes the approximate result need to satisfy user-defined requirements, such as error demand, response time. To meet the demand, [4] used stratified sampling to build multiple offline samples with different error bounds. Yan *et al.* proposed an error-bounded stratified sampling technique to minimize the sample size while satisfying a predefined error bound [8].

These sampling methods can also be extended to parallel and distributed systems to significantly reduce application execution time [32]. Combined with parallel frameworks like MapReduce, [6], [10], [31] respectively improved the

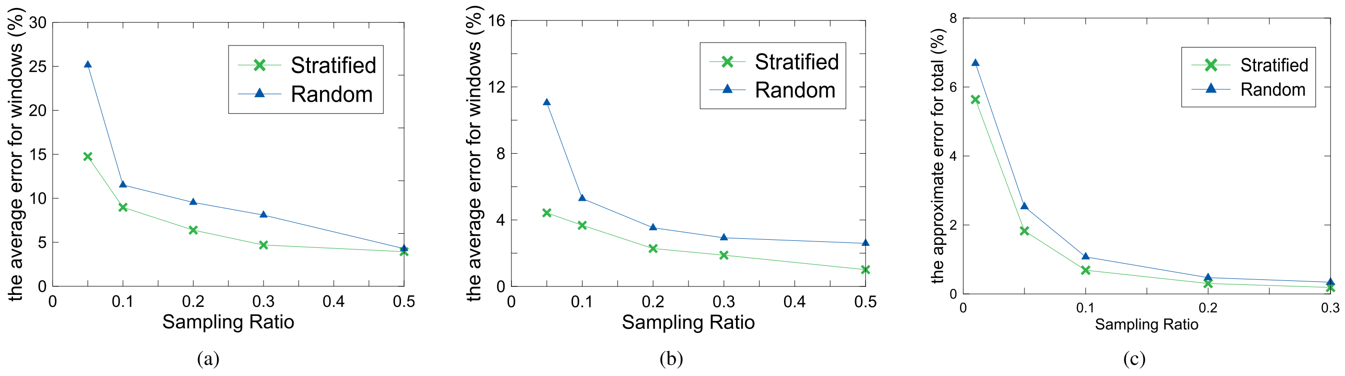


FIGURE 10. **(a)(b)(c) Average error comparison with random and stratified sampling schemes. (a) Window Size: 1000. (b) Window Size: 2000. (c) Error comparison for the total.**

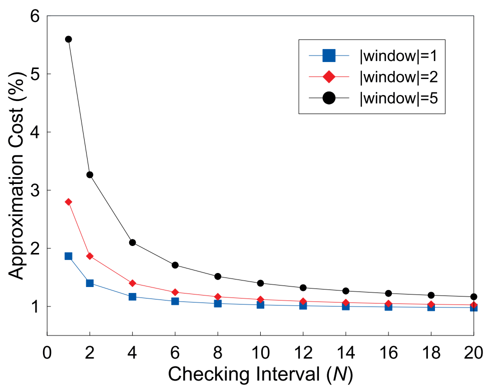


FIGURE 11. **Computational cost of approximation including error control with different checking intervals.**

processing performance for different data structures and application fields, including data mining, social networks. Based on the MapReduce framework, Goiri *et al.* [19] designed a prototype system, ApproxHadoop, to implement approximation-enabled applications through input data sampling and task dropping.

In recent years, more and more applications concentrate on streaming computing [33]–[35]. Different from static data, data streams continuously arrive at a high rate, which increases the difficulty of processing. On one hand, stream data may not be stored owing to storage limits, and we need to process them in a single pass; on the other hand, continuous arriving data brings about the problem that we cannot predict in advance its prior distribution information. Although the assumption is the future workload is similar to historical information, it is not always consistent with the real situation. The inconsistent prediction will produce ineffective samples [4].

In the early stage, Aggarwal [28] designed a temporal biased reservoir sampling for the data stream. The time-relevant weighted sampling was presented to find recent frequent items in [36]. They considered the effect of time during the stream evolution. Although these common sampling methods are available for streams, it is still inefficient without considering the context of stream data (e.g., data distribution). In recent works, Krishnan *et al.* [7] 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. Reference [37] described the stratified method can generate more representative samples with more accurate results. For online stream processing, [37] proposed two challenges: choosing the size of samples inside each stratum and the number of strata is difficult since the knowledge of data is unknown.

To overcome these limitations, we propose a tree-based online data learning strategy to divide the data items and allocate the weight for each stratum. In [8], Yan *et al.* proposed a stratified sampling technique to minimize the sample size while satisfying a predefined error bound. But they need to know the knowledge of data distribution and perform sorting, which may not be practical for online arriving data. To make an improvement, we design a hash mapping to stratify the arrived data to the corresponding strata.

To ensure the output quality, most sampling-based works tend to provide a theoretical error guarantee [11], [16]. However, the output quality may not be guaranteed and it is possible that estimated results are unsatisfactory for customers because of the probability of sampling. Then we focus on the problem of quality control for the real-time stream processing. It is more significant since data arriving without being stored increases the difficulty of error correction.

There exist a few quality management strategies when using approximate techniques from the system level [38], [39]. For instance, Rumba is an online quality management system applied in an approximate accelerator-based computing environment [9]. It employs continuous light-weight check to automatically detect and correct large approximation errors.

In our paper, we target the real-time stream processing and also propose a customized error control strategy to assure output quality. When users specify an output quality requirement, our proposed strategy can make a corresponding error detection operation to instantly correct unsatisfactory results.

VIII. CONCLUSION

In this paper, 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 continuously arriving data. According to the stratification result obtained from data learning, a dynamic sampling scheme is designed to make a self-adjusting computation. For different user requirements, a customized error control strategy is designed to detect approximate results so as to timely correct larger errors. Experiment results with real-world datasets show that our proposed approximate framework can be well applied to real-time stream processing environments to make efficient approximate processing with online quality control.

REFERENCES

[1] X. Wang, X. Wei, Y. Liu, and S. Gao, "On pricing approximate queries," *Inf. Sci.*, vol. 453, pp. 198–215, Jul. 2018.

[2] CAIDA. *CAIDA: Center for Applied Internet Data Analysis*. Accessed: Jun. 26, 2018. [Online]. Available: http://www.caida.org/home/

[3] T. Lu, G. Liu, W. Li, S. Chang, and W. Guo, "Distributed sampling rate allocation for data quality maximization in rechargeable sensor networks," *J. Netw. Comput. Appl.*, vol. 80, pp. 1–9, Feb. 2017.

[4] 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 *Proc. ACM Eur. Conf. Comput. Syst.*, 2013, pp. 29–42.

[5] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically accurate program transformations," in *Proc. Int. Static Anal. Symp.* Berlin, Germany: Springer, 2011, pp. 316–333.

[6] A. Haque, S. Chandra, L. Khan, and C. Aggarwal, "Distributed adaptive importance sampling on graphical models using mapreduce," in *Proc. IEEE Int. Conf. Big Data*, Oct. 2014, pp. 597–602.

[7] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues, "IncApprox: A data analytics system for incremental approximate computing," in *Proc. Int. Conf. World Wide Web*, 2016, pp. 1133–1144.

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

[9] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *Proc. ACM/IEEE 42nd Int. Symp. Comput. Archit.*, Jun. 2016, pp. 554–566.

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

[11] S. L. Lohr, *Sampling: Design and Analysis*. Scarborough, ON, Canada, Nelson Education, 2009.

[12] RIP. (1988). *RIP: Routing Information Protocol*. Accessed: Feb. 22, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Routing_Information_Protocol

[13] S. Mallick, G. Hains, and C. S. Deme, "A resource prediction model for virtualization servers," in *Proc. Int. Conf. High Perform. Comput. Simulation*, Jul. 2012, pp. 667–671.

[14] X. Wang, X. Wei, S. Gao, Y. Liu, and Z. Li, "A novel auction-based query pricing schema," in *International Journal of Parallel Programming*. New York, NY, USA: Springer, 2017, pp. 1–22.

[15] M. Al-Kateb and B. S. Lee, "Stratified reservoir sampling over heterogeneous data streams," in *Proc. Int. Conf. Sci. Stat. Database Manage.*, 2010, pp. 621–639.

[16] W. Hoeffding, "Probability inequalities for sums of bounded random variables," *J. Amer. Stat. Assoc.*, vol. 58, no. 301, pp. 13–30, 1963.

[17] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 35–50, 2014.

[18] G. Ananthanarayanan *et al.*, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. OSDI*, Oct. 2010, vol. 10, no. 1, p. 24.

[19] Í. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing approximations to mapreduce frameworks," in *Proc. ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2015, pp. 383–397.

[20] Wikimedia. (2016). *Wikimedia*. Accessed: Feb. 28, 2018. [Online]. Available: https://dumps.wikimedia.org/enwiki

[21] Wikipedia. (2016). *Wikipedia: Wikipedia Database*. Accessed: Feb. 28, 2018. [Online]. Available: http://en.wikipedia.org/wiki/Wikipedia_database

[22] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*. New York, NY, USA: ACM, 2003, pp. 40–51.

[23] D. L. Quoc *et al.*, "ApproxJoin: Approximate distributed joins," in *Proc. ACM Symp. Cloud Comput.* New York, NY, USA: ACM, 2018, pp. 426–438.

[24] A. Pavan, K. Tangwongsan, S. Tirthapura, and K. L. Wu, "Counting and sampling triangles from a graph stream," *Proc. VLDB Endowment*, vol. 6, no. 14, pp. 1870–1881, 2013.

[25] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1449–1463.

[26] L. Wang, R. Christensen, F. Li, and K. Yi, "Spatial online sampling and aggregation," *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 84–95, 2015.

[27] D. Miao, X. Liu, and J. Li, "On the complexity of sampling query feedback restricted database repair of functional dependency violations," *Theor. Comput. Sci.*, vol. 609, pp. 594–605, Jan. 2016.

[28] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Proc. 32nd Int. Conf. Very Large Data Bases*, 2006, pp. 607–618.

[29] M. Al-Kateb and B. S. Lee, "Adaptive stratified reservoir sampling over heterogeneous data streams," *Inf. Syst.*, vol. 39, no. 1, pp. 199–216, Jan. 2014.

[30] X. Wang, R. T. B. Ma, Y. Xu, and Z. Li, "Sampling online social networks via heterogeneous statistics," in *Proc. IEEE Conf. Comput. Commun.*, Apr./May 2015, pp. 2587–2595.

[31] R. Levin and Y. Kanza, "Stratified-sampling over social networks using mapreduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2014, pp. 863–874.

[32] P. Triantafillou, "Towards intelligent distributed data systems for scalable efficient and accurate analytics," in *Proc. IEEE 38th Int. Conf. Distrib. Comput. Syst.*, Jul. 2018, pp. 1192–1202.

[33] M. Vlachou-Konchylaki, "Efficient data stream sampling on apache flink," M.S. thesis, School Comput. Sci. Commun., KTH Roy. Inst. Technol., Stockholm, Sweden, 2016.

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

[35] I. Manousakis, Í. Goiri, R. Bianchini, S. Rigo, and T. D. Nguyen, "Uncertainty propagation in data processing systems," in *Proc. ACM Symp. Cloud Comput.*, 2018, pp. 95–106.

[36] Y. Lim, J. Choi, and U. Kang, "Fast, accurate, and space-efficient tracking of time-weighted frequent items from data streams," in *Proc. 23rd ACM Int. Conf. Conf. Inf. Knowl. Manage.*, 2014, pp. 1109–1118.

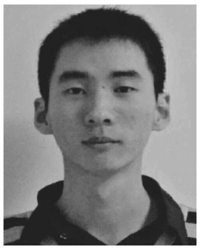
[37] R. El Sibai, Y. Chabchoub, J. Demerjian, Z. Kazi-Aoul, and K. Barbar, "Sampling algorithms in data stream environments," in *Proc. Int. Conf. Digit. Economy (ICDEc)*, Apr. 2016, pp. 29–36.

[38] M. Samadi and S. Mahlke, "CPU-GPU collaboration for output quality monitoring," in *Proc. 1st Workshop Approx. Comput. Across Syst. Stack*, 2014, pp. 1–3.

[39] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, pp. 62-1–62-33, Mar. 2016.

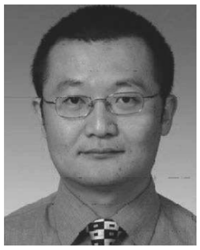
XIAOHUI WEI is currently a Professor and the Dean of the College of Computer Science and Technology, Jilin University, and also the Director of the High Performance Computing Center. His current research interests include resource scheduling for large distributed systems, infrastructure level virtualization, large-scale data processing systems, and fault-tolerant computing.

YUANYUAN LIU received the B.S. and M.A. degrees from the College of Computer Science and Technology, Jilin University, Changchun, Jilin, China, in 2013 and 2016, respectively, where she is currently pursuing the Ph.D. degree. Her research interests include approximate computing and big data analysis.



XINGWANG WANG received the B.Sc. degree from the College of Computer Science and Technology, Jilin University, Changchun, Jilin, China, in 2013, where he is currently pursuing the Ph.D. degree. His research interests include approximate computing and cloud computing.

SHANG GAO received the B.Math. degree (Hons.) from the University of Waterloo, Canada, in 2006, and the M.S. and Ph.D. degrees from the University of Calgary, Canada, in 2009 and 2014, respectively, all in computer science. From 2006 to 2007, he was an Engineer with SAP Labs China, Shanghai. Since 2014, he has been an Associate Professor with the College of Computer Science and Technology, Jilin University, China. His research interests include algorithmic game theory, cloud computing, and data systems.



LEI CHEN received the B.S. degree in computer science and engineering from Tianjin University, China, in 1994, the M.A. degree from the Asian Institute of Technology, Thailand, in 1997, and the Ph.D. degree in computer science from the University of Waterloo, Canada, in 2005. He is currently a Professor with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology. His research interests include crowdsourcing on social networks, uncertain and probabilistic databases, web data management, multimedia and time series databases, and privacy.