Online Anomaly Prediction for Robust Cluster Systems

Xiaohui Gu

North Carolina State University
Raleigh, NC 27695
gu@csc.ncsu.edu

Haixun Wang

IBM T. J. Watson Research Center
Hawthorne, NY 10532
haixun@us.ibm.com

***Abstract*—In this paper, we present a stream-based mining algorithm for online anomaly prediction. Many real-world applications such as data stream analysis requires continuous cluster operation. Unfortunately, today's large-scale cluster systems are still vulnerable to various software and hardware problems. System administrators are often overwhelmed by the tasks of correcting various system anomalies such as processing bottlenecks (i.e., full stream buffers), resource hot spots, and service level objective (SLO) violations. Our anomaly prediction scheme raises early alerts for impending system anomalies and suggests possible anomaly causes. Specifically, we employ Bayesian classification methods to capture different anomaly symptoms and infer anomaly causes. Markov models are introduced to capture the changing patterns of different measurement metrics. More importantly, our scheme combines Markov models and Bayesian classification methods to predict when a system anomaly will appear in the foreseeable future and what are the possible anomaly causes. To the best of our knowledge, our work provides the first stream-based mining algorithm for predicting system anomalies. We have implemented our approach within the IBM System S distributed stream processing cluster, and conducted case study experiments using fully implemented distributed data analysis applications processing real application workloads. Our experiments show that our approach efficiently predicts and diagnoses several bottleneck anomalies with high accuracy while imposing low overhead to the cluster system.**

I. Introduction

Modern computer systems, especially distributed systems, have become more and more complicated. This complexity inevitably makes the system management a challenging task. Many emerging applications, such as data stream processing applications [14], [12], [3], [15], requires 24x7 continuous system operation. Unfortunately, today's large-scale cluster systems are still vulnerable to various software and hardware problems that can cause system operation anomalies such as processing bottlenecks (i.e., full stream buffers) and service level objective (SLO) violations (e.g., response time $>$ 500 ms). System administrators are often overwhelmed by the tasks of correcting system anomalies under time pressure. In practice, many anomalies are still manually detected and recovered, which can cause long service downtime. Thus, it is imperative to provide automatic anomaly prediction and correction for large-scale distributed systems to achieve continuous system operation.

Anomaly prediction for complex systems is a challenging task. To raise advance anomaly alerts, we need to continuously monitor various system components using a set of metrics. Given a training dataset of such measures, we may be able to build a model to classify the current status of a system into two states: normal or abnormal. However, the real challenge lies in *classifying future data*, that is, data we have not seen yet, so that we can take preventive actions to steer the system away from the impending anomaly situation. Furthermore, in addition to be able to foresee the anomaly, we must also be able to find out the probable causes of the anomaly, so that we know what preventive actions to take. In this work, we introduce a novel approach to *classify future data for anomaly prediction*. Although anomaly detection (e.g., [4]) has been studied before, little research has addressed the anomaly prediction problem, that is, giving the probability that a certain type of anomaly will appear before the system enters the anomaly state. Anomaly prediction requires the system to perform online continuous classification on future data, which motivates us to design a new stream-based mining algorithm.

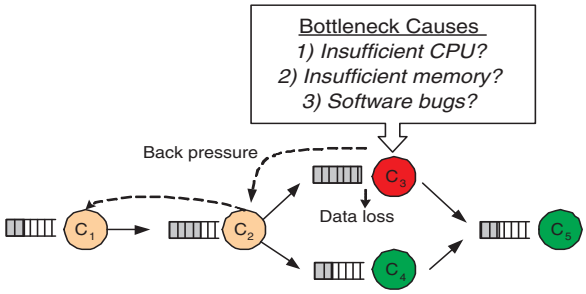

Fig. 1. Bottleneck anomaly in data stream processing system.

In this work, we focus on predicting the bottleneck anomaly, the most common anomaly in data stream processing clusters. A data stream processing application typically consists of a set of processing components. Each component accepts input data from its upstream component(s) and produces output data for its downstream component(s). A bottleneck appears in the distributed application when the input queue of a component reaches its upper limit. For example, in Figure 1, the component C_3 becomes the bottleneck in the stream processing application. Bottlenecks occur due to various reasons, such as i) a processing component is given

1084-4627/09 $25.00 © 2009 IEEE
DOI 10.1109/ICDE.2009.128

insufficient resources, ii) the input data rate exceeds the processing capacity of the component, or iii) the component contains software bugs (e.g., memory leak, buffer management error). The negative impact of a bottleneck component includes the dropping of application data since its input buffer cannot accept new data. If the bottleneck is not alleviated in time, the processing of the whole application will be stalled and suffer from data loss.

Traditionally, bottleneck alleviation is achieved by load shedding [25] (i.e., dropping a subset of input data to reduce input workload) or load distribution [32], [**?**] (i.e., distributing workload among a set of replicated components). However, those existing approaches have several limitations. First, they are carried out without knowing the underlying reasons that cause the bottleneck. For example, if the bottleneck is caused by insufficient CPU, allocating more memory or disk resource cannot alleviate the bottleneck. If the bottleneck is caused by bugs such as memory leak in component software, neither load shedding nor load distribution will help. Second, previous work performs bottleneck alleviation after a bottleneck occurs. If the bottleneck is not detected in time, the system may suffer from damages such as losing important application data. To address those problems, we propose a new bottleneck anomaly prediction solution that can raise alert for an impending bottleneck anomaly, estimate bottleneck pending time, and provide possible bottleneck causes.

In this paper, we present a new stream-based mining algorithm to achieve online anomaly prediction. Different from anomaly detection, anomaly prediction requires us to perform continuous classification on future data. Our stream mining algorithm integrates naive Bayesian classification method and Markov models to achieve the anomaly prediction goal. Our system continuously collects a set of runtime system-level and application-level measurements (e.g., available CPU and memory on a host, CPU and memory consumption of a component, input/output data rates of a component). We use those measurement streams to train a set of Bayesian classifiers to capture the *distinct symptoms* of different bottlenecks caused by various reasons (e.g., insufficient CPU, insufficient memory, memory leak bug). Our approach is based on the observation that bottlenecks caused by different reasons exhibit different anomaly symptoms. For example, the symptom of a bottleneck component caused by insufficient CPU (i.e., increasing CPU consumption) is different from the symptom of a bottleneck component caused by the memory leak bug (i.e., increasing memory consumption).

To achieve classification on future data, we employ Markov models to capture the changing patterns of different measurement metrics that are used as features by the Bayesian classifiers. In this study, we use discrete-time Markov-chains with a finite number of states. For continuous values, we discretize them into finite number of bins. Through Markov-chains, we predict the values of each metric for the next k time units. The Bayesian classifier is then used to predict the probability of different anomaly symptoms by combining the metric values. Thus, our online anomaly prediction system can generate an anomaly alert such as "A bottleneck anomaly caused by the reason X will appear at the component C_i after T time units with a probability P". The system can take a proper alleviation actions based on the bottleneck alert.

We have implemented the online anomaly prediction system within System S, a data stream management system developed at IBM which runs on a commercial cluster consisting of about 250 blade servers. We conduct experiments using a fully implemented distributed data analysis application processing real application workloads running on the cluster system [11]. Our implementation experience and experiment results reveal several interesting findings: 1) albeit its simplicity, naive Bayesian classifier can achieve high accuracy (i.e., up to 95% detection rate and 10-20 % false alarm rate) for a range of bottleneck symptoms caused by common reasons such as insufficient CPU/memory resources and internal software bugs (e.g., memory leak); 2) the Markov models can be combined with the naive Bayesian classifier to efficiently predict future bottleneck incidents and give rather accurate bottleneck pending time estimation (i.e., up to 80% accuracy); and 3) our analysis components are light-weight, whose training time is within several hundreds of milliseconds and whose online analysis time is less than 100 microseconds. Although our system implementation is still at its preliminary stage, the experiments show that our approach is promising for performing online anomaly prediction in cluster systems.

The paper is organized as follows. Section II presents the problem and gives an overview of our approach. Section III presents stream-based mining algorithms for online anomaly prediction. Section IV presents the prototype implementation and experimental results. Section V compares our work with related work. The paper concludes in Section VI.

II. Preliminary

In this section, we formulate the problem and provide an overview of our approach.

A. Problem Formulation

To achieve informed bottleneck prediction, the system needs to identify the underlying reason that causes the bottleneck. Examining a single metric such as queue length is insufficient. To see this, consider a bottleneck caused by insufficient CPU or insufficient memory. In both cases, the queue length may gradually increase until it reaches the maximum capacity. Consequently, it is impossible to identify the bottleneck cause by examining the queue length alone. However, there are often other metrics that can differentiate the causes of bottlenecks. In this case, available free memory and available CPU time are two such metrics. Therefore, we want to build a bottleneck classifier that incorporates multiple metrics called features that can collectively capture the distinctive symptoms of different bottlenecks.

As a toy example, Figure 2 shows a two-dimensional feature space where anomaly symptoms of three different causes (bottlenecks caused by insufficient CPU, insufficient memory, or both insufficient CPU and insufficient memory) form three

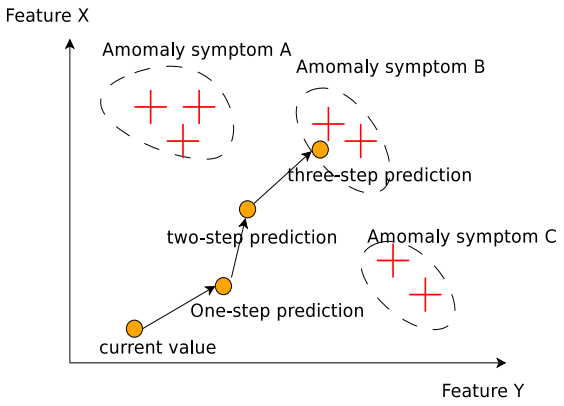

Fig. 2. Online anomaly prediction.

clusters. If we have enough labeled data in this feature space, we can learn a model to classify unlabeled points in the feature space.

However, in order to foresee bottlenecks, we need to apply the classifier on future data. Thus, the first task is to predict the future data. In Figure 2, we predict the position of a point in the feature space in one, two and three time steps[1]. As it shows in the figure, one possible outcome is that, in three time steps, the measurement point falls into the cluster representing anomaly symptom B. If that outcome has a large probability, the system should raise alert that an anomaly with symptom B will occur after three time steps. To predict feature values, the system needs to model the statistical changing patterns of different feature values. Combining both anomaly symptom classification and feature value prediction, the system can perform online anomaly prediction, that is, performing anomaly symptom classification on future data.

B. Overview of Our Approach

First, we learn different anomaly symptoms from historical data (*training data*), which consists of records of a fixed set of attributes. For system monitoring, the feature space $\mathbf{X}$ consists of a set of system-level and application-level measurements. Table I shows the feature metrics collected in our system. We consider both i) host-level metrics such as available memory, free CPU time, and free disk space, and ii) component-level metrics such as input data rate, output data rate, data processing time, and component memory usage values. We train naive Bayesian classifiers from the data, because i) a Bayesian classifier can be trained very efficiently, and ii) it produces posterior probabilities that can be combined with feature predictions to perform predictive anomaly classification.

The classifier enables us to tell whether data x indicates an anomaly situation. However, the goal is tell whether the system will have bottleneck situation in the future. Since the data of the future is not available, we need to predict future data before we apply the classifier on the predicted data. To this end, we employ Markov models to capture feature transition patterns.

[1]Each step represents a certain time interval, say 10 seconds.

Host Metrics	Description
AVAILCPU	percentage of free CPU cycles
FREEMEM	available memory
PAGEIN	virtual page in rate
PAGEOUT	virtual page out rate
MYFREEDISK	free disk space
Component Metrics	**Description**
RXSDOS	num. of received data objects
TXSDOS	num. of transmitted data objects
DPSDOS	num. of dropped data objects
RXBYTES	num. of received bytes
TXBYTES	num. of transmitted bytes
UTIME	process time spent in user mode.
STIME	process time spent in kernel mode
ROUTING	system data handling time
VMSIZE	address space used by a component
VMLCK	VM locked by the component
VMRSS	VM resident set size
VMDATA	VM usage of the heap
VMSTK	VM usage of the stack
VMEXE	VM executable
VMLIB	VM libraries

TABLE I
MONITORING METRICS.

This is indicated by Figure 3. Markov models have been used extensively in many fields to model stochastic processes [20]. In this study, we model each feature using one discrete-time Markov-chain of a finite number of states, where each state represents a feature value (continuous values are discretized into finite number of bins). The result of Markov simulation is a region in the feature space, where each point in the region is associated with a value, indicating the probability reaching that feature point.

Finally, we apply Bayesian classifier over data in the region. This requires us to compute posterior probability for every point in the region, and then find the expected probability. To reduce computation complexity, we rely on the assumption that feature values are independent. Although the assumption is naive, it has been shown that naive Bayesian classifier is very effective in many situations.

III. System Design

In this section, we present the design details of our stream-based mining algorithms for online anomaly prediction. We first describe a Bayesian classification approach to learning different anomaly symptoms. Second, we present a scheme of using discrete-time Markov chain models to capture the changing patterns of different metric values. Finally, we describe how to combine the above two schemes to predict the cause and pending time for an anomaly.

A. Anomaly Learning

Given system measurements at the current time, our goal is to find out whether and when an anomaly condition will

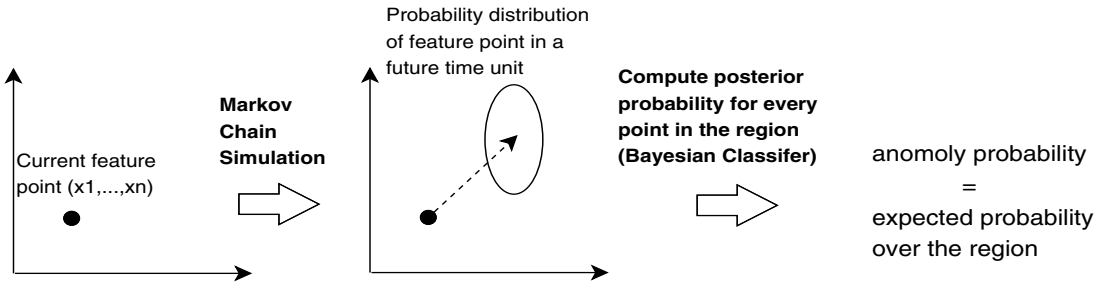

Fig. 3. overview of our approach

occur in the foreseeable future. For each type of anomaly, the training dataset consists of records in the form of $(\mathbf{x}, c)$, where $\mathbf{x}$ is a vector of system measurements, and $c = Yes/No$ indicates whether $\mathbf{x}$ represents an anomaly condition. Thus, the model learned from the training dataset only enables us to predict whether the current data $\mathbf{x}$ represents an anomaly condition, instead of whether an anomaly will occur in the future.

To find out whether an anomaly condition will occur in a future time unit, let us assume we know $\mathbf{X}$, the probability density function of $\mathbf{x}$ in that future time unit:

$$\mathbf{X} \sim p(\mathbf{x}) \tag{1}$$

In Section III-B, we describe how to derive $\mathbf{X}$, the probability density. For now, the question we want to answer is, given we know $p(\mathbf{x})$, how to predict whether an anomaly will occur in that future time unit.

Note that $p(\mathbf{x})$ is different from the estimated prior distribution $p(\mathbf{x}|\mathcal{D})$ that can be obtained from the training data $\mathcal{D}$. We consider each observation in $\mathcal{D}$ as an independent sample from an unknown distribution, while we are able to obtain $p(\mathbf{x})$ through some estimation mechanism. In this case, we take advantage of the temporal locality of the data and predict their feature values in the future time units from their current values [7].

Expected Classification: Assume we have a classifier that outputs the posterior probabilities of anomaly/normal, i.e., it outputs $P(C = anomaly|\mathbf{x})$ and $P(C = normal|\mathbf{x})$ for a give $\mathbf{x}$. Eq (1) gives $p(\mathbf{x})$, the distribution of feature values in a future time, with which we compute the expected logarithmic posterior probabilities:

$$E_{\mathbf{X}}(\log P(C = c|\mathbf{x})) = \int_{\mathbf{x}} (\log P(C = c|\mathbf{x}))p(\mathbf{x})d\mathbf{x} \tag{2}$$

We can thus make prediction about the future state. That is, we predict *anomaly* if

$$E_{\mathbf{X}}(\log P(anomaly|\mathbf{x})) \geq E_{\mathbf{X}}(\log P(normal|\mathbf{x}))$$

A natural question is how good the prediction is? Since we are classifying unseen data, so the concern about prediction quality comes more from our uncertainty about the future data rather than from the quality of the classifier itself. In other words, even if we have an oracle classifier, or a classifier which is always 100% accurate, we may still have low quality prediction if we have a very rough estimation of $\mathbf{x}$.

In order to measure the certainty of our prediction, we simply compare the expected logarithmic posterior probabilities [7] for *anomaly* and *normal*:

$$\delta = E_{\mathbf{X}}(\log P(anomaly|\mathbf{x})) - E_{\mathbf{X}}(\log P(normal|\mathbf{x})) \tag{3}$$

Clearly, the value of $|\delta|$ indicates the confidence of our prediction: the larger the $|\delta|$, the more confident we are about our prediction (either *anomaly* or *normal*). Eq 3 is known as the Q1 measure in [7]. For our study, we raise an alert of future *anomaly* if $\delta \geq d$, where d is a constant value that represents the confidence threshold of the alert.

We need to choose a reasonable value of d. In system monitoring, *anomaly* is a rare event: Most of the time the system is *normal* state. In other words, we have

$$\delta_0 = \log P(anomaly) - \log P(normal) < 0 \tag{4}$$

It means, if $\mathbf{x}$ covers a large feature space[2], then only a small region in that feature space represents *anomaly*. Value δ_0 is the prior difference of the likelihood, and usually $\delta_0 < 0$. If δ, the expected difference in the future is larger than δ_0, then we have reason to believe that the system is less healthy than it normally is. Thus, we set $d = \delta_0$, or, we raise an alert if

$$\delta \geq \delta_0 \tag{5}$$

that is, when the difference between anomaly and normal likelihood in a future time unit is more significant than indicated by their prior differences.

Naive Bayesian Classifier: The above reasoning is intuitive and easy to understand. However, it can be computationally challenging: according to Eq 2, in order to compute $E_{\mathbf{X}}(\log P(C = c|\mathbf{x}))$, we need to evaluate $P(C = c|\mathbf{x})$ for every possible $\mathbf{x}$ in the multi-dimensional feature space. If the dimensionality is high, the computation will be very costly or even infeasible.

To solve this problem, we make a naive assumption: each metric is conditionally independent given the class labels. With this assumption, a very simple classifier, the naive Bayesian classifier, can be applied. In spite of its naive assumption, it has been shown in many studies that the performance of naive

[2]that is, $p(\mathbf{x}) > 0$ for $\mathbf{x}$ in a large feature space

Bayesian classifiers are competitive with other sophisticated classifiers (such as decision trees, nearest-neighbor methods, etc.) for a large variety of data sets [10], [16].

With naive Bayesian classifier, we have

$$E_{\mathbf{X}}(\log P(c|\mathbf{x})) = E_{\mathbf{X}}\left(\log \frac{P(\mathbf{x}|c)P(c)}{\sum_c P(\mathbf{x}|c)P(c)}\right) \quad (6)$$

Once we plug it into Eq 3, the denominator $\sum_c P(\mathbf{x}|c)P(c)$ will disappear in the log ratio. In other words, whether an alert will be raised or not only depends on the relative value. So we ignore the denominator and derive the following.

$$\begin{aligned} E_{\mathbf{X}}\left[\log(P(\mathbf{x}|c)P(c))\right] &= E_{\mathbf{X}} \log P(\mathbf{x}|c) + E_{\mathbf{X}} \log P(c) \\ &= E_{\mathbf{X}} \sum_i \log P(x_i|c) + \log P(c) \\ &= \sum_i E_{\mathbf{X}} \log P(x_i|c) + \log P(c) \\ &= \sum_i E_{X_i} \log P(x_i|c) + \log P(c) \end{aligned}$$

Thus, instead of having to compute $E_{\mathbf{X}}(\log P(c|\mathbf{x}))$, we only need to compute $E_{X_i}(\log P(x_i|c))$, that is, instead of relying on the joint density function $\mathbf{X} \sim p(\mathbf{x})$, we only rely on the distribution of each feature $X_j \sim p(x_j)$, which is much easier to obtain, and makes Eq 2 feasible to compute [7].

B. Evolving Feature Model

In Section III-A, we assumed that we know the feature distribution in a future time unit. Here, we discuss how to derive the future feature distribution.

Consider any system metric x. We discretize its values into M bins by equi-depth discretization. The reason we use equi-depth discretization is that some system metrics have outlier values, which makes traditional equi-width discretization suboptimal. We then build a Markov-chain for that system metric, that is, we learn the state transition matrix P_x for system metric x. Assume we know the feature value at time t_0: $x = s_i,\ 1 \leq i \leq M$. The distribution of the feature value at t_0 is simply $p_0(x) = e_i$, where e_i is a $1 \times M$ unit row vector with 1 at position i and 0's at other positions. The distribution of the feature value in the next time unit t_1 is $p_1(x) = p_0(x)P_x = e_iP_x$. In the next time unit t_2, the distribution of the feature value becomes $p_2(x) = p_1(x)P_x = e_iP_x^2$.

Thus, we have derived the feature value distribution of x for any time in the future: at time t_i, the distribution is $p_i(x)$. Clearly, when i becomes large, the distribution will converge to $p(x) = \pi$, where π is the prior distribution (among the historic data based on which we have built the Markov-chain) of the feature values. In other words, the probability of a certain feature value in the next time unit is approximately the fraction of its occurrence in the historic data. But, as the gap between the current time and the time when we last investigated the feature values becomes larger, the temporal correlation will disappear.

Now, in order to answer the question whether and when an anomaly condition will occur in the foreseeable future, we only need to plug in $p_i(x), \forall i$ into Eq 3. An alert is raised for time t if the feature distributions at time t makes $\delta > d$. In our experiments, we analyze the performance of the alert.

An important issue in system monitoring is that due to changing workload, any anomaly diagnose mechanism must take into consideration the time-varying class distribution of anomaly and normal states. The topic of classifying time-varying data streams has been well explored (e.g., [29], [30], [5]). However, we need to apply classifiers on unseen future data instead of current data. To this end, we adopt a finite-memory Markov-chain model and incrementally update its parameters so that they reflect the characteristics of the most recent data. The main idea is to maintain the Markov-chains using a sliding window of the most recent W transitions and update the parameters of the Markov-chains when new observations are available [7].

In this study, we assume the Markov-chains for different system metrics are independent. That is, given the values of a system metric at current time, the distribution of its feature values in the next time unit is independent of the distributions of other system metrics. This assumption makes it easier for us to solve the problem numerically by using, e.g., Monte Carlo methods. More specifically, with the independence assumption, we can draw samples for each feature following its own distribution, independent of other features. Without this assumption, we have to use some special Monte Carlo sampling techniques [17], [31]. To study the cases in which the feature distributions are dependent is among our future work.

C. Algorithms

In this section, we explain the Bayesian learning algorithm and the online alert generation algorithm more formally. We also discuss how to judge the quality of the alerts generated by our algorithms.

Bayesian Learning: Algorithm 1 is invoked periodically to train a Bayesian classifier for each anomaly type. In other words, it induces a set of binary classifiers $\{C_1, \cdots, C_k\}$ so that for each unlabeled sample $\mathbf{x}$, C_i will make a binary decision of whether or not $\mathbf{x}$ is a case of anomaly type i.

Algorithm 1 simply computes the frequency of anomaly and normal cases for each attribute value (after equi-depth discretization). However, in a small training dataset, we may find certain attribute values having zero frequency: $p(x_i = j|c) = 0$. A testing sample with that feature value will always have zero posterior probability according to the Bayesian rule. To alleviate this problem, we assumes there are m imaginary cases whose feature values have equal probability of being in any bin (m-estimate). The likelihood probabilities using m-estimate is then computed as on line 11.

Online Alert Generation: Algorithm 2 implements the online alert system. It takes system measures generated at equally spaced time interval (for example, each interval is 3 seconds), and decides if an alert should be raised. Algorithm 2 returns an integer value s to indicate that the next anomaly is likely to occur after $s \geq 1$ time units in the future. If the return

Algorithm 1 Bayesian Learning Algorithm

inputs: $\mathcal{D}$: a dataset with class labels
inputs: bin_i: number of equi-depth bins for attribute i
inputs: m: m-estimator
outputs: logarithmic prior and likelihood probabilities

initialize all counters in $count$ and $count_i$ to 0
for all sample $(\mathbf{x}, c)$ in $\mathcal{D}$ **do**
  discretize $\mathbf{x}$ by equi-depth binning, where the boundaries of bins are learned in a separate pass on the data before Bayesian learning is started
  $count[c] \leftarrow count[c] + 1$
  for all feature i in $\mathbf{x}$ **do**
    $count_i[c][x_i] \leftarrow count_i[c][x_i] + 1$
  end for
end for
$p(c) \leftarrow log\frac{count[c]}{\sum_c count[c]}$ for $c \in$ {fail,normal}
for all value j of feature i **do**
  $p(x_i = j|c) \leftarrow log\frac{count_i[c][x_i]+m/bin_i}{count[c]+m}$
end for
return $p(c)$ and $p(x_i = j|c)$, $\forall c, i, j$

Algorithm 2 Online Alert Algorithm

inputs: $\mathbf{x}^1, \cdots, \mathbf{x}^k, \cdots$: a stream of system measurements
inputs: N: number of future time units covered by the alert
outputs: alert

update Markov-chains' state transition matrices for each $(\mathbf{x}^k, \mathbf{x}^{k+1})$ pair
$s \leftarrow 1$
while $s < N$ **do**
  compute $p(x_i^k)$, the value distribution of the i-th feature in the s^{th} future time unit, based on the current value x_i^k and P_i^s, where P_i is the state transition matrix for feature i
  compute expected logarithmic posterior $E_{\mathbf{x}}(\log P(c|\mathbf{x}))$ using Eq 6
  compute δ for time unit s using Eq 3
  if $\delta \geq \delta_0$ **then**
    return s
  end if
  $s \leftarrow s + 1$
end while
return 0

value is 0, it means no anomaly is predicted to happen within up to N time units in the future. Note that for presentation simplicity, the algorithm shown above is for detecting one single anomaly type. However, it is straightforward to modify it to provide alert for all anomaly types.

Quality of Alerts: In order to evaluate the quality of the alerts generated by Algorithm 2, we must know when real anomalies happen in the future. We run Algorithm 2 on a labeled dataset, and compare the relative positions of alerts and real anomaly occurrences.

At any time, we decide if an alert should be raised to indicate that there will be an anomaly s time units away. Assume the real next anomaly is f time units aways. Since

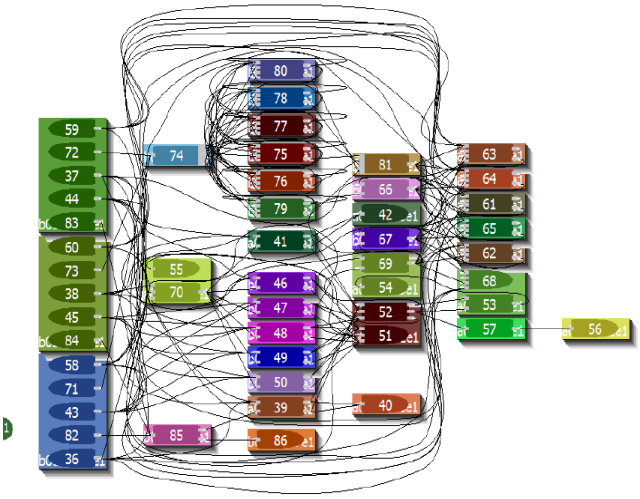

Fig. 4. Case study distributed stream processing application.

we only evaluate the next N steps, we have $s < N$, and also we only consider $f < 2N$ (anomalies further into the future have little predictability). We consider several cases.

1) Next anomaly is within N time units; no alert is generated, or an alert is generated after the anomaly, that is, $f < s$. This is a *false negative* case, as we either fail to raise an alert for an imminent anomaly, or fail to raise an alert early enough.
2) No anomaly within N time units; no alert is generated. This is apparently a *true negative* case.
3) Alert is generated; no anomaly within $2N$ time units. This is a *false positive* case. We take unnecessary premonition.
4) Alert is generated; next anomaly occurs after the generated alert, that is $s < f$. This is a *true positive* case.

The true/false positive/negative statistics enable us to compute detection rates and false alarm rates. Note that because the purpose of generating alerts is to enable us to take premonition, we do not insist that the predicted anomaly and the real anomaly coincide at same time unit. Instead, as long as the real anomaly occurs after the alert (within 2N), we consider it as a true positive classification. However, the distance between the alert and the real anomaly is also an important indication of the quality of the alert. As we want the distance be as short as possible. Our experiments present both detection rate, false alarm rate, and the distance statistics.

IV. System Evaluation

A. Implementation

We have implemented the online anomaly prediction system within IBM System S [15], a large-scale data stream processing system running on a commercial cluster consisting of 250 blade servers. Each server host has dual Intel Xeon 3.2GHZ CPUs and 2 to 4 GB memory. All of our experiments

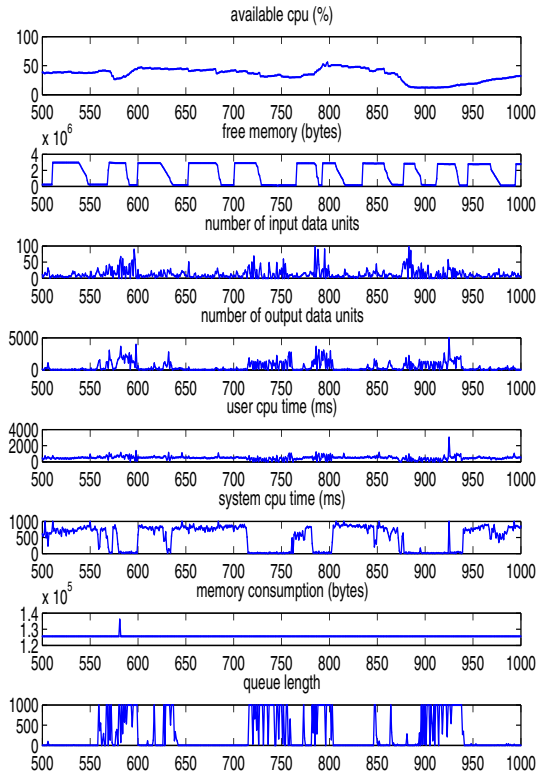

Fig. 5. Metric values of several bottleneck anomaly occurrences.

are conducted on the cluster system. We test our system using DAC (Disaster Assistance Claim monitoring), a fully implemented data stream processing applications processing real application workloads. The DAC application consists of 51 fully implemented software components running on 35 hosts, illustrated by Figure **??**. DAC is a multi-modal stream analytic and monitoring application. Inspired by the observation that various kinds of frauds seem to have always been committed against any disaster assistance program, DAC aims to identify, in real time before money is dispensed, (a) the processed claims that are fraudulent or unfairly treated and (b) the problematic agents and their accomplices engaged in illegal activities. The input workloads consist of various real application traces such as voice-over-IP conversations, email logs, news feeds, and news videos. Although we use data stream applications in our experiments, our approach is applicable to many other distributed applications such as workflow processing, multi-tier enterprise applications, and composite web services.

Metric Stream Collection: To achieve self-learning system management, we deploy monitoring components on distributed hosts to collect runtime distributed system metrics. To achieve generality, we strive to avoid complicated application instrumentation to acquire detailed measurement data. Instead, we try to use those features that can be easily collected through operating systems and middleware infrastructure. Table I in Section II-B lists the major metrics collected in our system. The host-level metrics (e.g., available CPU, available memory) are acquired by querying the OS interface (e.g., the `/proc` interface). The component-specific metrics (e.g., input/output data rates, CPU/memory consumptions) are collected via the stream processing middleware infrastructure. For example, the middleware acquires the input/output data rates of a component by sniffing the input/output data traffic going through the component's input/output ports. The middleware also maintains a mapping from the component identifier to the process identifier so that it can acquire the resource consumption information of the component by querying the `/proc` interface. Thus, the online anomaly prediction system can easily acquire the component-level metrics from the middleware system without requiring extra application instrumentation. Figure 4 illustrates the runtime values of a partial set of the metrics for a component that experiences a bottleneck problem caused by insufficient memory. We observe that the bottleneck exists during time periods of [550, 650], [720,800], and [900, 950]. From those metric values, online anomaly prediction learns both normal component behavior and different bottleneck symptoms. To achieve low-overhead metric monitoring, our system leverages its own bottleneck prediction capability to achieve adaptive metric sampling. For each monitored component whose state is predicted as normal, we use a low sampling rate since normal samples are mostly redundant. If the component is predicted to become a bottleneck in a foreseeable future, the system increases the sampling rate to collect more precise measurements since failure samples are much more rarer than normal samples.

Data Logging: The online anomaly prediction system selectively logs a subset of received metric measurements that will be used as the training data for updating classifiers. Before original measurement data can be used as training data, we need to annotate the measurement data with proper labels. The system can label each measurement samples as "bottleneck-positive" or "bottleneck-negative" based on the queue length of the component. If the queue length has reached its upper-limit, the component is considered to become a bottleneck in the distributed application. To distinguish real bottleneck problem from transient load spikes, we may want to delay the failure labelling slightly. The system labels the measurement data as bottleneck-positive if it sees the component's input queue is full for several consecutive time units. In particular, we use majority voting over the last W time instants. Therefore, when a new set of measurements arrives, we estimate the corresponding label. In order to decide whether to label the measurement as bottleneck-positive at time t, we examine the queue length $Q_{t-W+1}, \ldots, Q_{t-1}, \hat{\ell}_t$. If more than $W/2$ of those queue lengths are full, we label the measurement data as bottleneck-positive.

To diagnose the bottleneck causes, the classifiers also need the system to provide some measurements annotated with

different reasons that cause the bottleneck. Thus, we can train different classifiers to capture the symptoms of different bottlenecks. Essentially, we rely on human examination to acquire accurate bottleneck cause labels. The system administrators or application developers can retrieve the log data annotated with bottleneck-negative or bottleneck-positive labels. He or she can diagnose the bottleneck causes and add the bottleneck cause labels into the log data. In addition, we can leverage previous metric attribution [8] and history clustering techniques [9] to acquire the bottleneck cause label more easily.

Decentralized system architecture: We implemented the online anomaly prediction system using a fully distributed architecture to achieve scalability. The system consists of a set of metric monitoring components and measurement analysis components. We run one monitoring daemon process at each distributed host in the cluster to collect the measurements for the host and all the components running on the host. We create one analysis component (i.e., predictive bottleneck classifier) to continuously predict and diagnose bottleneck failure for each application component. If one application component has multiple similar replicas, we can employ one analysis component to track the status of the replica group. The analysis components are the major resource consumers in our system. To achieve low-overhead system management, online anomaly prediction strives to exploit idle cluster resources for performing data collection and analysis tasks. We first decouples the monitoring and analysis tasks. Thus, we can run the analysis task on a different host from the monitoring task that is typically required to be co-located with the monitored component. Second, we implement analysis component migration mechanism to dynamically place them on the lightly-loaded hosts with most abundant idle resources. To avoid imposing extra overhead to the system, we adopt an opportunistic algorithm to perform dynamic analysis component placement. Each host can acquire the load conditions about a set of hosts by "sniffing" the measurement metrics collected by its local analysis component. When the host discovers other hosts that have more idle resources than itself, it can offload some of its analysis components to more lightly-loaded hosts.

B. Experiment Setup

In our experiments, we deploy the online anomaly prediction system in our cluster system by running a monitoring daemon process on each host and dynamically creating analysis components for all running application components. We test our system using the DAC application [11] described in Section IV-A. The input workloads used by the application are real application data by replaying a set of trace files such as wide-area TCP traffic flows taken from the Internet Traffic Archive [1] and news video streams taken from NIST TRECVID dataset [2]. In our experiments, each application run lasts about 5000 seconds. We inject bottleneck faults in some components at different time instants to test whether our system can raise alert for component bottlenecks with high accuracy. We will show the prediction results for a join component[3]. The join component continuously correlates data from different streams using a pre-defined join condition. For video streams, the join predicate is to find similar video scenes among different news videos. For network traffic streams, the join predicate is to find network packets with common source and destination IP addresses.

As we mentioned before, component bottlenecks can be caused by different reasons. In our experiments, we test our system on four bottleneck causes: 1) *insufficient CPU*: we start a CPU-intensive component on the same host to compete the CPU resource with the monitored component; 2) *insufficient memory*: we start a memory-bound component on the same host to grab memory resource from the monitored component; 3) *insufficient CPU & insufficient memory*: we start both CPU-bound and memory-bound background workloads to make the monitored component become the bottleneck in the application; and 4) *component software bug*: the monitored component contains a memory leak bug where its memory consumption accumulates as the component executes the buggy code segment that keep forgetting to free memory. We raise bottleneck alarms based on the criterion explained in Section III. Each measurement in the testing dataset is annotated with its true labels. By comparing the predictive classification results and true labels, we calculate the number of true positives $N_{\rm tp}$ where a bottleneck failure happens after the predicted time interval, the number of true negatives $N_{\rm tn}$ where no bottleneck happens and no alarm is raised, the number of false-positives $N_{\rm fp}$ where a bottleneck happen before the system raises any alarm, and the number of $N_{\rm fn}$ where an alarm is raised but no bottleneck happens after the predicted time interval. Following the standard definition, we can calculate the true positive rate A_{tp} and false positive A_{fp} as follows,

$$A_{tp} = \frac{N_{\rm tp}}{N_{\rm tp} + N_{\rm fn}}, A_{fp} = \frac{N_{\rm fp}}{N_{\rm fp} + N_{\rm tn}} \quad (7)$$

C. Results and Analysis

Figure 5-8 shows the predictive bottleneck classification accuracy achieved by our system for the bottleneck join components caused by lacking sufficient CPU resource, lacking sufficient memory resource, lacking both CPU and memory resources, and a memory leak bug, respectively. In our case study distributed application, there are six replicated join components that run on different hosts. Those join components perform the same operation on similar input workload, which exhibit similar but not identical behavior. In our experiments, we inject different faults (e.g., insufficient CPU, insufficient memory, insufficient CPU and memory, and memory leak bug) at different time instants to make those join components become processing bottlenecks in the studied distributed application. We use the log data of one join component to train the Bayesian classifier and Markov models. We then use the induced Bayesian classifier and Markov models to predict and diagnose the bottleneck failure of the other five components.

[3]We also conduct experiments on other types of components. The results show similar trend, which are omitted here due to space limitation.

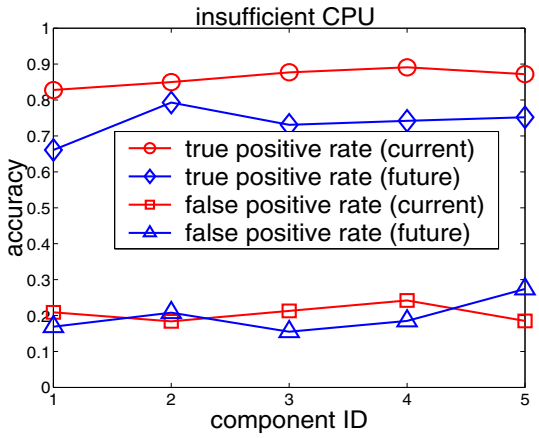

Fig. 6. Predictive classification accuracy for insufficient CPU bottlenecks.

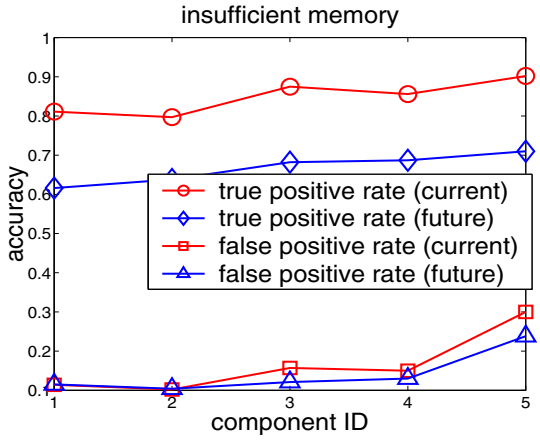

Fig. 7. Predictive classification accuracy for insufficient memory bottlenecks.

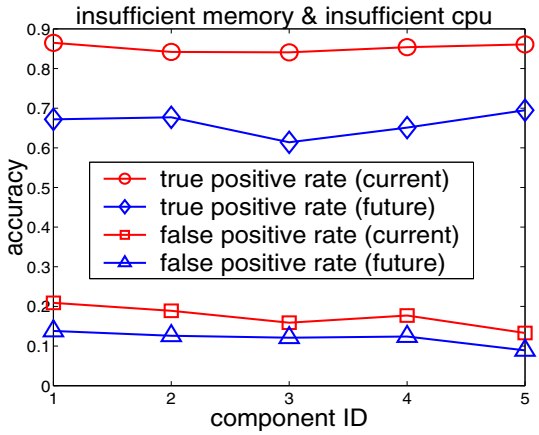

Fig. 8. Predictive classification accuracy for insufficient Cpu and memory bottlenecks.

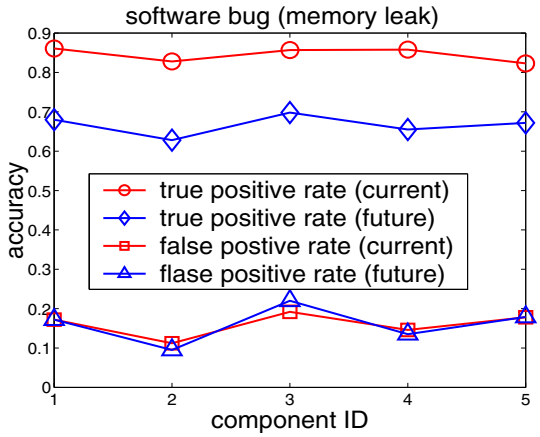

Fig. 9. Predictive classification accuracy for memory leak bottlenecks.

When we classify whether the current component behavior exhibits insufficient CPU bottleneck symptom, we only need to invoke the Bayesian classifier. Thus, the two curves marked with "true positive rate (current)" and "false positive rate (current)" show the Bayesian classification accuracy. Overall, the results show that naive Bayesian classification can achieve good classification accuracy for different bottleneck symptoms.

We then combine the Markov models and Bayesian classifier to perform predictive bottleneck classification for future measurement data. In this set of experiments, the look-ahead window includes 10 intervals and each interval is 3 seconds. In our experiments, the default metric sampling rate is one sample per three seconds. In order words, the analysis component takes the current values of all monitored metrics and predicts their future values in the next 10 time intervals using the Markov models. We then classify the state of those predictive feature values using the Bayesian classifier. We can derive a final classification probability (Equation 3) and decide whether to raise alert based on Equation 5. If any of the predicted future measurements within the lookahead window gives a positive classification result, the analysis component will produce a positive predictive classification result.

In Figure 5-8, the two curves, marked with "true positive rate (future)" and "false positive rate (future)", show the classification accuracy for future metric values. Overall, our system can still achieve reasonably good classification accuracy for future metric values. We observe that the classifier generally has lower true positive rate for future values than for current values. The reason is that even if the classifier issues an alarm for an impending bottleneck but the bottleneck happens before the predicted time instant, we still consider that the classifier makes a false negative error. We also observe that the classifier can have lower false positive rate for future measurements than for current measurements. The reason is that if the classifier predicts that a bottleneck will happen at a future time instant t, we consider the classifier gives correct prediction if the bottleneck appears anytime within the lookahead window after the time t.

We now evaluate the system's performance on estimating the bottleneck pending time for different bottleneck symptoms caused by four different reasons, illustrated by Figures 9-

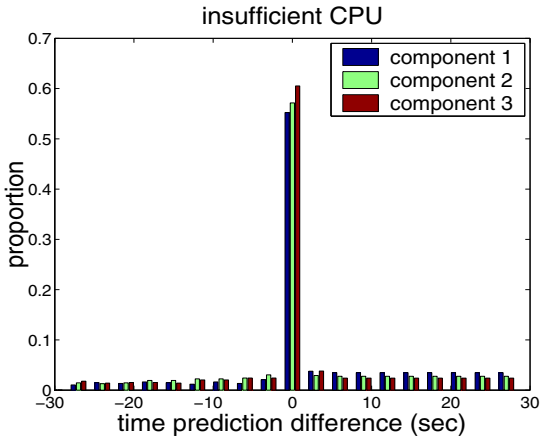

Fig. 10. Bottleneck pending time estimation for insufficient Cpu bottlenecks.

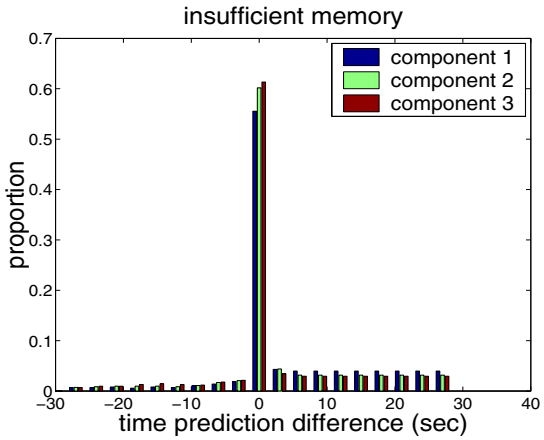

Fig. 11. Bottleneck pending time estimation for insufficient memory bottlenecks.

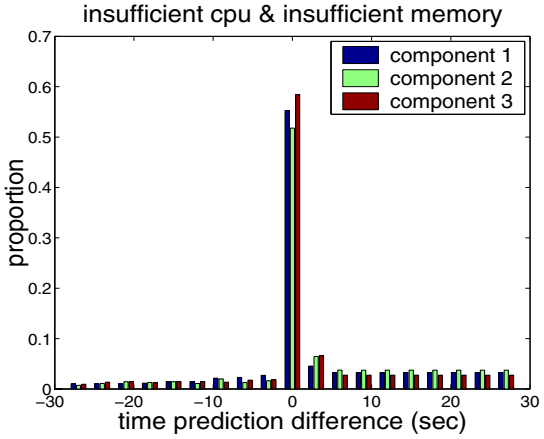

Fig. 12. Bottleneck pending time estimation for insufficient Cpu and memory bottlenecks.

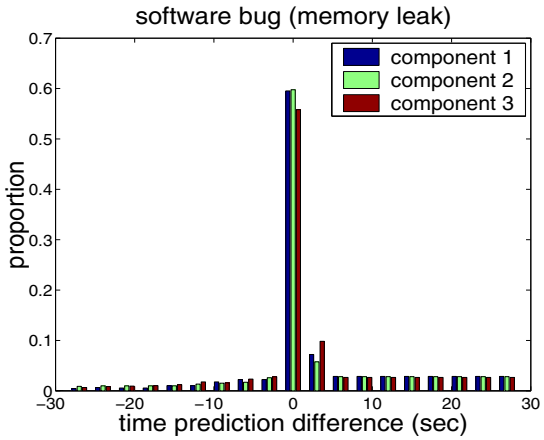

Fig. 13. Bottleneck pending time estimation for memory leak bottlenecks.

12. We calculate the time difference between the predicted bottleneck time and true bottleneck time. For example, if the system predicts that a bottleneck symptom will appear at the $k'th$ time interval within the lookahead window and the real bottleneck happens at the $m'th$ time interval, the time difference is calculated by $m - k$. If the time difference is zero, it means that the bottleneck appears in the exact time interval predicted by the system. Similar to the previous set of experiments, the look-ahead window includes 10 intervals and each interval is 3 seconds. In Figures 9-12, the X-axis shows the time difference and the Y-axis shows the proportion of the bottleneck predictions within the time difference bucket. We observe that the system can give accurate bottleneck pending time estimation in most cases. Moreover, we prefer making positive time difference mistakes to making negative time difference mistakes since it has less negative impact that a bottleneck happens after the predicted time interval.

We also evaluate the overhead of the online anomaly prediction system. Figure 13 shows the cumulative distribution of mean training time collected in different experiment runs. The training time includes both Markov model training time and naive Bayesian classifier training time. We observe that the total training time is within several hundreds of milliseconds. Figure 14 shows the cumulative distribution of mean prediction time (including both markov prediction time and Bayesian classification time) collected in different experiment runs. The results show that our prediction algorithm is fast, which requires tens of micro-seconds. Compared to the measurement collection period that is typically more than one second, our classification time is well-qualified for performing bottleneck failure prediction. Using biased sampling, a large-scale distributed application like our case study distributed applications with 51 components running on 35 hosts consumes about 240MB storage space to log its execution time for 24 hours, which is relatively small compared to the storage capacity of modern cluster server hosts.

V. Related Work

We apply classifiers on a future data distribution instead of on current data. A related work in the data mining domain is load shedding in classifying data streams [7], [6], where resources (e.g., CPU time) are directed to analyzing data that

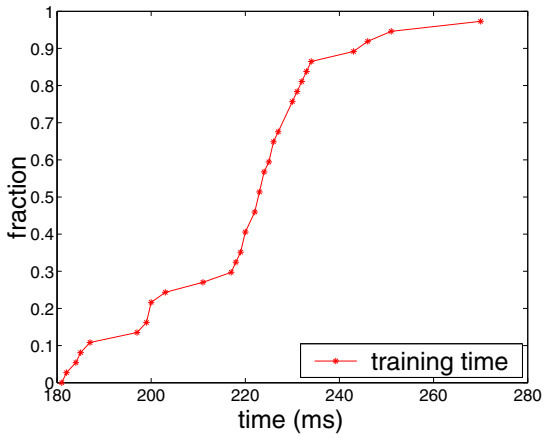

Fig. 14. Anomaly prediction model training time.

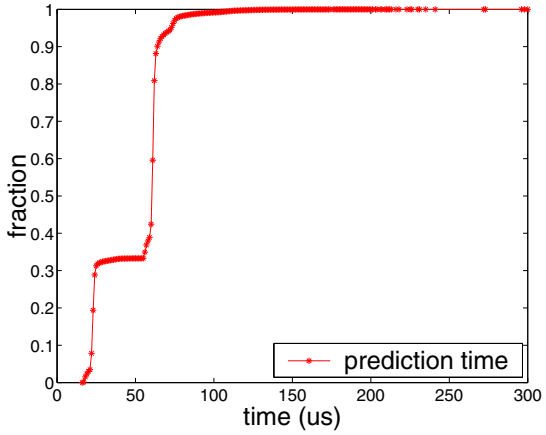

Fig. 15. Online prediction time.

can improve the quality of classification. Since the goal is to avoid analysing unnecessary data, predicting data distribution is required. However, in anomaly prevention, we are more interested in knowing when the system is going to enter an unfavorable state, and whether the gap is long enough to make preventive actions. In [**?**], we have presented an initial design of the online anomaly prediction system, which adapts decision tree classifiers for online anomaly prediction. However, classification alone cannot provide time-to-anomaly estimation. In contrast, this work combines metric value prediction with state classification, which can not only predict how likely an anomaly symptom will appear, but also when the predicted anomaly will appear.

Recently, statistical machine learning methods are used for autonomic system management. SMART [21] studied different nonparametric statistical approaches for predicting disk failures. Since prediction errors of disk failure has significant penalty, SMART focuses on feature selection so as to avoid false-alarms as much as possible. Cohen et al. proposed to apply Tree Augmented Bayesian Networks to correlate system-level metrics with high-level performance metrics for performance diagnosis [8]. Parekh et al. compared different classification methods to discover bottleneck metrics in the configuration of multi-tier enterprise applications [23]. Zhang et al. proposed to use ensembles of models for diagnosing performance problems [33]. Cohen et al. proposed a clustering approach to capture the essential characteristic of a system state for problem diagnosis [9]. Powers et al. explored different statistical multi-variate methods to perform performance prediction for enterprise system [24]. Gniady et al. proposed program-counter-based classification schemes to optimize buffer caching [13]. Mirza et al. proposed machine learning based algorithms to predict TCP throughput [19]. Mesnier et al. proposed a new relative fitness model for modelling the performance of storage devices using regression trees [18]. Different from the above work, our research focuses on developing *stream-based mining algorithm* for *online* anomaly prediction. To the best of our knowledge, it is the first stream-based mining algorithm that combines feature value prediction and anomaly symptom classification for online system anomaly prediction.

Our work is also related to performance monitoring and cluster management work. Stardust is a fine-grained system instrumentation framework that can collect end-to-end traces of requests in distributed systems and provide query interface for performance metrics [26]. Software rejuvenation is a proactive failure management technique that uses Stochastic Reward Nets to model and analyze cluster systems, which can periodically stops a running software, cleans its internal state, and restarts it to prevent unexpected failures due to software aging [27]. Our system is complementary to the above work and can benefit from previous performance monitoring techniques. We can also combine the online anomaly prediction scheme with the software rejuvenation technique to alleviate the processing bottlenecks in distributed applications. For example, if our system predicts a bottleneck will appear in the distributed application and the bottleneck is caused by software aging, we can invoke the software rejuvenation to alleviate the bottleneck.

VI. Conclusion

In many applications such as robust control of complicated systems, it is essential to raise alert in advance so that the system can steer away from impending disasters or failures. This requires us to mine data that has not arrived yet. In our work, we derive a distribution of the future data based on the current data, and instead of classify the data, we classify the distribution. To the best of our knowledge, this work makes the first attempt to apply statistical machine learning methods on predicting bottleneck anomalies in distributed data stream processing systems. We have tested our system using fully implemented distributed data analysis applications processing real application workloads. Our experiments show that online anomaly prediction can predict and diagnose a range of bottleneck anomaly symptoms with high accuracy. Our system is feasible for large-scale cluster systems and imposes low-overhead to the system.

VII. Acknowledgement

We thank Nagui Halim, the principal investigator of the System S project, and Chitra Venkatramani, Henrique Andrade, Yoonho Park, Philippe L. Selo, Kun-Lung Wu, Bugra Gedik, Lisa Amini for helping us to test the anomaly prediction algorithms on the System S cluster.

References

[1] Internet Traffic Archive. *http://ita.ee.lbl.gov/*.

[2] NIST TREC Video Archive. *http://www-nlpir.nist.gov/projects/trecvid/*.

[3] Daniel J. Abadi and et al. The Design of the Borealis Stream Processing Engine. *Proc. of CIDR*, 2005.

[4] John Breese and Russ Blake. Automating computer bottleneck detection with belief nets. In *Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 36–45, San Francisco, CA, 1995. Morgan Kaufmann.

[5] S. Chen, H. Wang, S. Zhou, and P. S Yu. Stop Chasing Trends: Discovering High Order Models in Evolving Data. *Proc. of IEEE International Conference on Data Engineering (ICDE)*, 2008.

[6] Y. Chi, H. Wang, and P. S. Yu. Loadstar: load shedding in data stream mining. In *Proc. of the 31st Intl. Conf. on Very Large Databases (VLDB)*, pages 1302–1305, 2005.

[7] Y. Chi, P. S. Yu, H. Wang, and R. Muntz. Loadstar: A load shedding scheme for classifying data streams. In *Proceedings of the 4th SIAM International Conference on Data Mining (SDM)*, 2005.

[8] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. *Proc. of OSDI*, 2004.

[9] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. *SOSP*, 2005.

[10] P. Domingos and M. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Mach. Learn.*, 29(2-3):103–130, 1997.

[11] K-L. Wu et al. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. *Proc. of VLDB*, 2007.

[12] S. Krishnamurthy et al. TelegraphCQ: An Architectural Status Report. *IEEE Data Engineering Bulletin, 26(1):11-18*, March 2003.

[13] C. Gniady, A. R. Butt, and Y. C. Hu. Program Counter Based Pattern Classification in Buffer Caching. *Proc. of OSDI*, 2004.

[14] The STREAM Group. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin, 26(1):19-26*, March 2003.

[15] X. Gu, S. Papadimitriou, P. S. Yu, and S.-P. Chang. Toward Predictive Failure Management for Distributed Stream Processing Systems. *Proc. of ICDCS*, 2008.

[16] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. *Proc. of ICDE*, 2007.

[17] N. Jain and et al. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. *Proc. of SIGMOD*, 2006.

[18] P. Langley, W. Iba, and K. Thompson. Analysis of Bayesian classifiers. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[19] J. S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer-Verlag, 2001.

[20] M. Mesnier, M. Wachs, R. R. Sambasivan, A. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. *Proc. of SIGMETRICS*, 2007.

[21] M. Mirza, J. Sommers, P. Barford, and J. Zhu. A Machine Learning Approach to TCP Throughput Prediction. *Proc. of ACM SIGMETRICS*, 2007.

[22] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[23] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado. Comparison of machine learning methods for predicting failures in hard drives. *Journal of Machine Learning Research*, 2005.

[24] Jason Parekh, Gueyoung Jung, Galen Swint, Calton Pu, and Akhil Sahai. Comparison of Performance Analysis Approaches for Bottleneck Detection in Multi-Tier Enterprise Applications. *Proc. of IWQOS*, 2006.

[25] R. Powers, I. Cohen, and M. Goldszmidt. Short term performance forecasting in enterprise systems. *ACM Knowledge Discovery and Data mining (KDD)*, 2005.

[26] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. *Proc. of VLDB*, 2003.

[27] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking Activity in a Distributed Storage System. *Proc. of SIGMETRICS*, 2006.

[28] K. Vaidyanathan, R. E. Harper, S. W. Hunter, and K. S. Trivedi. Analysis and Implementation of Software Rejuvenation in Cluster Systems. *Proc. of SIGMETRICS*, 2004.

[29] H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003.

[30] H. Wang, J. Yin, J. Pei, P. S. Yu, and J. X. Yu. Suppressing model overfitting in mining concept-drifting data streams. In *SIGKDD*, 2006.

[31] J. Xie, J. Yang, Y. Chen, H. Wang, and P. S. Yu. A Sampling-Based Approach to Information Recovery. In *ICDE*, pages 476–485, 2008.

[32] Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic Load Distribution in the Borealis Stream Processor. *Proc. of ICDE*, April 2005.

[33] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox. Ensemble of models for automated diagnosis of system performance problems. *Proc. of DSN*, 2005.