Online System Problem Detection by Mining Patterns of Console Logs

Wei Xu*, Ling Huang†, Armando Fox*, David Patterson*, Michael Jordan*

EECS Department, UC Berkeley, Berkeley, CA, USA

Email: {*xuw,fox,pattrsn,jordan*}*@cs.berkeley.edu*

†*Intel Labs Berkeley, Berkeley, CA, USA*

Email: *ling.huang@intel.com*

Abstract—**We describe a novel application of using data mining and statistical learning methods to automatically monitor and detect abnormal execution traces from console logs in an online setting. Different from existing solutions, we use a two stage detection system. The first stage uses frequent pattern mining and distribution estimation techniques to capture the dominant patterns (both frequent sequences and time duration). The second stage use principal component analysis based anomaly detection technique to identify actual problems. Using real system data from a 203-node Hadoop [1] cluster, we show that we can not only achieve highly accurate and fast problem detection, but also help operators better understand execution patterns in their system.**

I. MOTIVATION AND OVERVIEW

Internet services today often run in data centers consisting of thousands of servers. At these scales, non-failstop "performance failures" are common and may even indicate serious impending failures. Operators would therefore like to be notified of such problems quickly. Despite the existence of a variety of monitoring tools, the monitoring already available in every application is often ignored: the humble console log.

Console logs are convenient to use (only *printf* is required) and reflect the developers' original ideas about what events are valuable to report, including errors, execution tracing, or statistics about the program's internal state. But exploiting this information is difficult because console logs are both machine-unfriendly (they usually consist of unstructured text messages with no obvious schema or structure) and human-unfriendly (each developer logs information useful for her own debugging, but large systems consist of many software modules developed by different people, and log messages from different modules must often be correlated to identify a problem). These challenges make it difficult for operators to understand log messages, let alone analyze them usefully in an online setting.

Our previous work has shown promise in applying statistical machine learning and information retrieval to the problem of console log analysis [28]. Specifically, source code analysis recovers structure from console logs, and anomaly detection techniques infer which sets of messages may indicate operational problems. However, that work presents an offline algorithm that examines the entire log of a multi-day operational session, whereas operators need to be informed of such problems as they occur.

To this end, our first contribution in this paper is a novel two-stage *online* log processing approach that combines frequent pattern mining with Principal Component Analysis (PCA) based anomaly detection for system runtime problem detection. In particular, we show how to trade off time-to-detection vs. accuracy in the online setting by augmenting frequent-sequence information with timestamp information. Our method is completely automatic, and tuning its input parameters requires no special knowledge of the machine learning techniques we use. Our technique is general: [28] surveyed 22 systems, most of which have logs that are amenable to the approach described in this paper. As a beneficial side effect, the pattern mining aspect of our approach can potentially help operators better understand system behavior even under normal conditions.

Our second contribution is an empirical re-evaluation of our technique on the same labeled dataset used in [28]: 24 million lines of free-text logs from a 48-hour run of a production open-source application, the Hadoop File System (HDFS) [1] , running on a 203-machine cluster. We successfully identify anomalous conditions indicative of operational problems; in nearly all respects, we match or exceed the detection accuracy of the offline approach with small detection latencies that make our approach suitable for online use.

The rest of the paper proceeds as follows. In Section II we review related work. Section III reviews some elements of the offline approach of [28] that we also use in our online approach, described in section IV. We show experimental results in Section VII and comment on the limitations of our approach and other noteworthy aspects of the results in Section VIII, concluding in Section IX.

II. RELATED WORK

Online log analysis. The typical tools used by operators to analyze console logs, security audit logs, etc. [20], [8], [18] usually require rules (e.g., regular expressions to match logs) to be manually written and maintained as software changes. In contrast, we discover the "rules" automatically.

Sisyphus [23] provides some online analysis based on relative ratio of different terms in logs. In order to counter random interleavings in logs, it has to aggregate logs from a long period (tens of minutes).

[9], [17] use time-series analysis techniques to model the time intervals of periodical events. These patterns model the long term trend of event periodicity, while we monitor traces on individual events.

Path-based analysis. Chen, Kıcıman et al. used clustering [2] and probabilistic context free grammars [3] to analyze execution paths in server systems by manually instrumenting software components. X-Trace [6] now provides a framework for cross-layer path-based instrumentation collection across a distributed system. We collect the trace information from console logs rather than instrumenting the application, so we must deal with noisier data than would be produced by instrumentation, but we believe our analysis techniques would apply to path data too.

Using data mining techniques to solve computer system problems. Within the vigorous research area of frequent pattern mining [7], we are particularly interested in sequential pattern mining techniques, which mine frequently occurring ordered subsequences as patterns. For example, Generalized Sequential Patterns (GSP) [22] is a representative Apriori-based algorithm; SPADE [31] is a vertical format-based mining method; PrefixSpan [19] is a pattern-growth approach to sequential pattern mining. We extend the techniques to address the unique challenges of our problem described in Section V.

Frequent pattern mining techniques have also been used to analyze words in messages to understand the structure of console logs [24], [25] and to discover recurring runtime execution patterns in the Linux kernel [14]. Our frequent pattern analysis focuses on anomaly detection.

Data mining has been widely used for profiling end-hosts and networks [27], [12], detecting anomalous events and other intrusions [13], [30], detecting system and software bugs [15], [32], as well as configuration problems [26]. These techniques analyze aggregate data while we use traces from individual operations.

[23] uses information theory to find the words in logs most likely to indicate actual problems. In contrast, we consider message traces and detect anomalies of such traces. [29] models sequence patterns with a mixture of Hidden Markov Model (HMMs) from the original log traces. Due to the interleaving nature of log messages, the model is very complex.

III. CONSOLE LOG PREPROCESSING

In this section we review some log preprocessing techniques described in [28] that we also use in this paper. The necessary preprocessing steps are as follows: 1) parsing the logs to recover the inherent structure in the log messages; 2) reconstructing *traces* (execution sequences of related events) by automatically discovering which messages relate to the same program object (data block, filename, etc.) and putting them in order; and 3) constructing numerically-valued *feature vectors* from these traces that can be subjected to PCA-based anomaly detection.

Log parsing. The method presented in [28] can eliminate most of the ad-hoc guessing in parsing free text logs. The method first analyzes the source code of the program generating the console log to discover the "schemas" of all log messages. Specifically, it examines the printing statements in the source code (e.g. `printf`) and perform type analysis on the arguments to these statements. The technique distinguishes the parts of each message that are constant strings (the *message type*) from the parts that refer to identifiers such as program variables or program objects (the *message variables*) with high accuracy. The parsing technique is stateless, so it is easy to implement it in a data stream processor for our online setting.

Following conventions in system management work [9], we use the term *event* to refer to the data structure containing the elements of a parsed log message. Specifically, we define an *event* to be a tuple consisting of a timestamp, the event type (the message type of the parsed log message), and a list of variable fields referred to in the parsed log message (message variables). In an online setting, the streaming console log becomes an *event stream* after the parsing step.

Identifying event traces. Our detection technique relies on analyzing *traces*, which are sets of events related to the same program object. For example, a set of messages referring to the opening, positioning, writing, and closing of the same file would constitute an event trace for that file. Within the event stream, however, events of different types and referring to different sets of variables are all interleaved. One way to extract traces from the stream is to *group by* certain field of the events, a typical operation for stream data processing. The challenge is how to automatically determine the grouping key. We use the method described in [28], which automatically finds the grouping key from historical log data by discovering which message variables correspond to *identifiers* of objects manipulated by the program. All events reporting the same identifier constitute an event trace for that identifier. This "group-by" process also occurs in a stream processor. With the grouping key discovered, we implemented a trivial "group-by" stream processor that converts the single interleaved event stream into many event traces, one for each identifier. In particular, we assume the event traces to be independent from each other.

Representing the event traces. Lastly, we need to convert the event traces to a numerical representation suitable for applying PCA detector, which occurs in the second phase of our approach. In [28], each whole event trace is represented by a *message count vector* (MCV), which has a structure analogous to the *bag of words* model in information retrieval, where the "document" is the group of messages in an event trace. The MCV is an N-dimensional vector where N is the size of the set of all useful message types across all

groups (analogous to all possible "terms"), and the value of vector element y_i is the number of times event i appears in a group (corresponding to "term frequency"). For example, in a system consisting of four event types, opening, reading, writing and closing, a trace of (opening, reading, closing) will be represented in MCV as (1,1,0,1) while (opening, writing, writing) can be represented as (1,0,2,0).

Message count vectors are a compact representation of event traces, but two problems preclude their direct use in our online scenario. First, they do not carry any time information, so they cannot be used to detect operations that are anomalous due to events being spaced too far apart in time (i.e. slowness). Second, the original MCVs are constructed based on the entire event traces which could span arbitrarily long time. As we show later, this is usually not possible in an online setting. We use the MCV to represent a *session*, which we define as a subset events in an event trace representing a single logical operation in the system and have predictable bound in its duration. We show in Section IV how sessions are automatically discovered.

IV. TWO-STAGE ONLINE ANOMALY DETECTION

Compared to offline approaches such as [28], the fundamental problem of online analysis is that we cannot see the complete event trace at once. For example, in offline detection, a trace may be marked as abnormal because an event is missing; e.g. if a write operation to a file fails, its trace may lack a "closing" message. In online analysis, there is no way to know (other than waiting until the end of the run) if the missing event will ever come, yet the whole point of online detection is to make an assessment in a timely manner. We emphasize that detection time is determined only by how long the algorithm has to wait before making a decision. The computation time of the detection algorithm is negligible compared to this wait.

Effective online detection therefore requires striking a balance between accuracy and time to detection. At one extreme, if we wait to see the entire trace before attempting any detection, our results should be as accurate as offline detection but with excessive time to detection. At the other extreme, if we try to make a determination of anomalous behavior as soon as a single event appears, we lose the ability to perform anomaly detection based on *patterns* (a group of related events), yet [28] shows that analyzing patterns rather than the individual events is key to accurate detection.

We make this tradeoff by designing a two-stage detection method. The first stage uses frequent pattern mining to capture the most common (i.e., normal) session, that is, those traces with a high support level. The patterns include both frequent-event set and time information. This information can be used to determine when a trace is "probably complete" and can be made available for anomaly detection. The second stage considers only non-pattern events that make it through the first stage, applying PCA-based anomaly
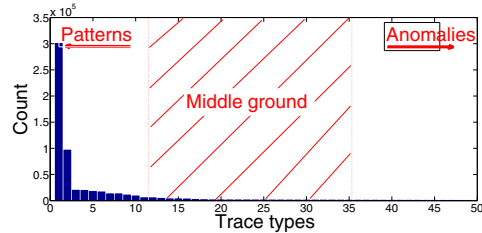

Figure 2. Histogram of 50 most frequent traces. Some traces are extremely frequent, and some are extremely rare, but there is a large "middle ground" which is neither pattern nor anomaly for sure.

detection to them. In each stage, we build a model based on archived history and update it periodically with new data, and use it for online detection. Both model estimation and online detection involves domain-specific considerations about console logs.

Figure 2 shows clearly why a two-stage approach is needed. The histogram of different event traces in our data shows that some traces clearly occur extremely frequently while others are extremely rare. It is reasonable to mark the dominant traces as "normal" behavior and the rare outliers as "anomalous", but this leaves a large middle ground of traces that are neither obviously dominant nor obviously anomalous. These traces in the middle ground are sometimes normal ones with added random noise such as interleavings. We want our detection method to tolerate the random noise. If we reduce the minimal support level to include more of these middle-ground cases, random noise (e.g. overlapping or incorrect ordering) will be introduced to the patterns, reducing the quality of the patterns.

Instead we pass the middle-ground cases to a PCA-based anomaly detector as non-pattern events. Since PCA is a statistical method that is able to match "inexact" patterns, it is more robust to random noise than the frequent-pattern mining used in stage 1 and can detect rare events among the middle-ground cases. Intuitively, the pattern-based method provides timely detection for the majority of events, minimizing the time to wait for the complete trace, while subsequent PCA-based detection handles the false alarms generated by the first stage and greatly improves detection accuracy. An additional benefit to the two-stage approach is that the frequent patterns from stage 1 can help operators to better understand the behavior of their systems and tune the detection to include domain-specific knowledge.

Although PCA is more robust against random noises than pattern mining and thus a suitable method for dealing with the noisy middle-ground events, frequent pattern mining has the advantage of being able to capture time information among events and providing an intuitive representation of dominant patterns. Our two-stage approach integrates the advantages of both methods. We now describe each stage in detail in the following two sections.
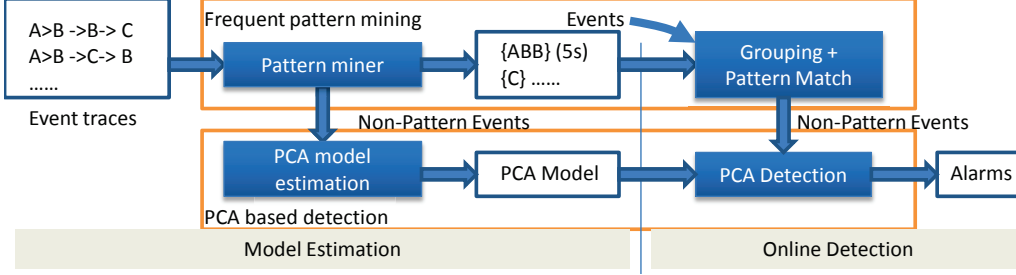
Figure 1. Overview of the two stage online detection systems.

V. STAGE 1: FREQUENT PATTERN MINING

As defined in Section III, an *event trace* is a group of events that reports the same identifier. We further define a *session* to be a subset of closely-related events in the same event trace that has a predictable duration. The *duration* of a session is the time difference between the earliest and latest timestamps of events in the session.

We define a *frequent pattern* to be a session and its duration distribution such that: 1) the session is frequent in many event traces; 2) most (e.g., 99.95th percentile) of the session's duration is less than T_{max}, a user-specified maximum allowable *detection latency* (the time between an event occurring and the decision of whether the event is normal or abnormal). Condition (1) guarantees that the pattern covers common cases so it is likely to be a normal behavior. Condition (2) guarantees the pattern can be detected in a short time. We mine the archived data periodically for frequent patterns. These patterns are used to filter out normal events in the online phase.

We cannot apply generic frequent sequence mining techniques because 1) sessions many interleave in the event traces (e.g. two reads happen at the same time) thus "transaction" boundaries are not clear. We need to simultaneously segment an event trace into sessions and mine patterns. However, because the durations of sessions can have large variations, fixed time windows will not give satisfactory segmentation, which suggest that we shall model the distribution of durations. 2) Events can be reordered in the traces because of unsynchronized clock in a distributed system, which preludes the use of techniques requiring total ordering of events. In our algorithm described below, we use frequent patterns to tolerate the poor time-based segmentation accuracy resulting from random session interleavings. The frequent patterns, once discovered, can be used to de-interleave the events to estimate a clean duration model.

A. Combining time and sequence information

Our novel approach combines time and event sequence information for accurate pattern detection using a 3-step iterative method. In a nutshell, we first use time information to (inaccurately) segment an event trace into sessions and then mine these inaccurate segments to identify the most frequent pattern. We then go back to the original data and

find out the actual time distribution of the sessions of the most frequent pattern. Finally we remove all events that match this frequent pattern from original data and iterate on the remaining data to find the next most frequent pattern.

1. Use time gaps to find first session in each execution trace (coarsely). In this step, for each execution trace, we first scan through each event until we find an event followed by a time gap more than 10 times the duration since the start of the execution sequence (the time gap size is a configurable parameter). We treat all events preceding the gap as a session (represented by an MCV). This segmentation can be very inaccurate: Due to interleaving sessions, irrelevant events might be included in the session and due to the randomness in session duration, events may be missing from the session. The inaccuracy is tolerated by the next step when finding most frequent patterns.

2. Identify the dominant session. We prefer a pattern that contains all events in a session. This is true in most cases due to the way sessions get segmented: with high probability (though not always), happening in a short time often indicates that the events represent a single logical operation, especially when the support level is high (recall the definition of sessions at the beginning of this section).

We use two criteria to select the dominant pattern. (1) We start with the medoid of all sessions considered (recall that the sessions are represented by MCVs). By definition, the medoid has the minimal aggregated distance from all other data points, which indicates that it is a good representative of all data points. Intuitively, a medoid is similar to the centroid (or mean) in the space, except that the medoid must be an actual data point. Criterion 1 guarantees that the selected dominant session is a *good representative* of the sessions examined. (2) We require the session to have a minimal support of $0.2M$ from all M event traces. If the medoid does not meet this minimal support, we choose the next closest session (data point) that does. Criterion 2 guarantees that the selected session is in fact dominant, in addition to being a good representative. The selection criteria are robust over a wide range of minimal support values because the normal traces are indeed in the majority in the log. In fact, in our experiments, various support values between $0.1M$ and $0.5M$ all resulted in the same selection results.

3. Refine result using the frequent session and compute

duration statistics. Notice that the pattern from step 2 is based on coarsely segmented sessions, and may not reflect the correct duration distribution of all sessions of that type. Now because we know the events we are expecting to complete a session, we can go back to the original data and find all events that match the frequent session and then estimate the duration distribution from the matching sessions (detailed in Section V-B). Using the duration distribution, we can compute a *cutoff time* T_{cut} (represents the time that most sessions of the pattern "should" complete) for the pattern as the ηth percentile of the distribution. We show in Section VII-B that this step significantly improved detection results. We also remove all matching events from the original traces, preparing the data for the next iteration. Notice that T_{cut} can be very long, due to large dispersion in durations in some operations. In the case that $T_{cut} > T_{max}$, the pattern is discarded and not used in the detection stage.

We then return to step 1 and iterate until no patterns with the minimal support level remain. Since Step 3 always removes something from the dataset, the iteration is guaranteed to terminate. The remaining events are used to construct the PCA model.

The dominant patterns are expected to be stable; however, in order to accommodate changes in the operation environment, we update patterns used in detector as a periodic (infrequent) offline process (i.e. the detector uses the patterns discovered but never update them online). In this way, we can both keep the online detector simple, and avoid poisoning the patterns with transient abnormal periods.

B. Estimating distributions of session durations

To enable timely online detection, we need to know how long any given pattern "should" take to complete. To this end, we estimate the distribution of session durations for each pattern. Based on this distribution, we compute the cutoff time T_{cut}, e.g., 99.95^{th} percentile of the distribution, for each pattern, after which most sessions of this pattern would complete.

To choose a distribution to fit our data, we observe that within each pattern, the histogram of session durations has both dominant values and fat tails, as shown by two examples (Patterns 1 and 2 in Table I in Section VII-A) in Figure 3 (a) and (b). Power-law distribution has been widely used to model data with long tails for its nice mathematical properties [5], [16]. We choose it to model our data, and a log-log plot confirms that the tails of our data approximately follow the power-law distribution (Figure 3 (c) and (d)).

To estimate the parameters of the distribution, we adopt the approach proposed in [4], which combines maximum-likelihood fitting methods with goodness-of-fit tests based on the Kolmogorov-Smirnov (KS) statistics [10]. In real applications, few datasets obey power-laws for all values. More often the power-law applies only to values greater than some minimum $x_{min} > 0$, i.e. to the tail of the distribution.
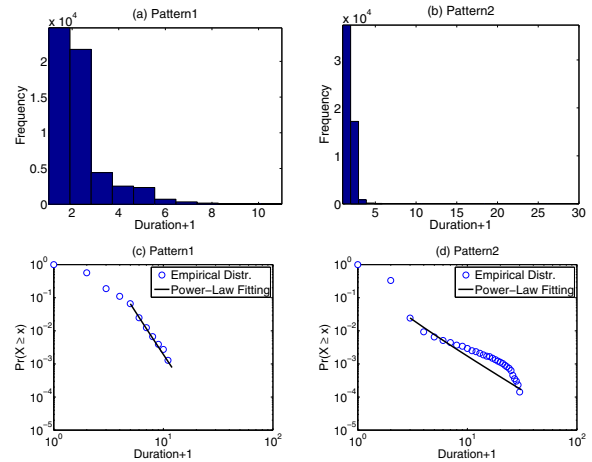

Figure 3.   Tail of durations follow power-law distribution.

For samples below this threshold, we use the histogram as its empirical distribution. So we essentially use a mixture distribution with two components to model the duration values: a power-law distribution for the tail (values above x_{min}), which has weight w, and a histogram for values below x_{min}, which has weight $(1 - w)$.

For durations that take only integer values, we consider the case with a probability distribution of the form $p(x) = Pr(X = x) = Cx^{-\beta}$. It is not difficult to show that the normalizing constant is given by:

$$C(\beta, x_{min}) = \left(\sum_{i=0}^{\infty} (i + x_{min})^{-\beta} \right)^{-1}. \quad (1)$$

Assuming x_{min} is known (the way for estimating x_{min} is discussed afterward), the Maximum Likelihood Estimator (MLE) of the scaling parameter β is approximately

$$\hat{\beta} \approx 1 + n \left[\sum_{i=1}^{n} \ln \frac{x_i}{x_{min} - 0.5} \right]^{-1}, \quad (2)$$

where $x_i, i = 1 \ldots n$ are the observed duration values that $x_i \geq x_{min}$.

To estimate x_{min}, we choose a value that makes the probability distributions of the measured data and the best-fit power-law model as similar as possible above x_{min}. We use KS statistics to measure the distance between two distributions, and estimate $\hat{x}_{min}$ as the value of x_{min} that minimizes the KS statistics between the empirical CDF of the data for the observations with value at least x_{min} and the fitted power-law model that best fits the data in the region with all $x_i \geq x_{min}$.

Figure 3 (c) and (d) show the empirical distributions (circles) and the fitted power-law models (solid lines) for patterns 1 and 2, respectively. With the model, the CDF $P_p(x) = Pr(X < x)$ of the power-law distribution is

$$P_p(x) = 1.0 - C(\beta, x)/C(\beta, x_{min}), \quad (3)$$

where $C(\beta, x)$ is defined in Eq. (1). Then, for $\eta \geq 1.0 - w$,

the ηth percentile of the mixture distribution is the value of x_η that satisfies the following equation:

$$P_p(x_\eta) = (\eta - (1.0 - w))/w, \qquad (4)$$

where $P_p(x)$ is defined in Eq. (3). We show the estimated 99.90^{th}, 99.95^{th} and 99.99^{th} percentiles of the mixture distributions of Patterns 1 and 2 and the improvements to the detection precision in Section VII.

C. Implementation of Stage 1

The pattern based detector receives the event stream from the log parser. If an event is part of some execution traces we are monitoring (because it contains an identifier), the detector groups it with other events with the same identifier and checks if any subset of the event group matches a frequent pattern. If a subset matches, all matching events are removed from the detector's memory (there might still be some non-matching events left in the queue). Removing matched events keeps the size of in-memory event history small and greatly improves the efficiency of the detector.

Logically, we try matching all event sets to all patterns. We used a naïve method that attempts each. We believe the naïve method is good enough in many systems because the number of patterns is usually small and the traces are short (because developers only log the most important stages on the execution path). However, in cases where many long patterns are used, we can use more advanced data structures such as suffix trees [21] to improve the matching efficiency.

If we do not find any matching pattern, the event is added to the queue with a timeout number T_o based on the event timestamp T. If the event matches one or more patterns, we choose the one with the largest cutoff time (T_{cut}) and set $T_o = T + T_{cut}$; if the event does not match any pattern (because the event is not frequent enough to be included in any pattern), we set $T_o = T + T_{max}$. Notice that because T_{cut} is usually much smaller than T_{max}, we can achieve fast detection on the majority of events.

The detector periodically checks all traces (currently the period is set to 1 second— this parameter has a small effect on detection time, but no effect on accuracy). When it finds events that have reached their timeout, it constructs their message count vectors (as described in Section III) and sends them to the second stage PCA-based detector.

The intuition behind this approach is that an event is passed through to the PCA-based detector as soon as we can be reasonably sure that it does not "belong to" any of the frequent patterns being monitored. We call these *non-pattern events*.

VI. STAGE 2: PCA DETECTION

The vectors representing the non-pattern events emitted from Stage 1 are significantly noisier than the frequent patterns. The noise comes from uncaptured interleaving, high variations in duration and the true anomalies. To uncover the true anomalies from this noisy data, we use a statistical anomaly detection method, the PCA detector, which is shown to be accurate in offline problem detection from console logs and from many other systems [28], [13].

As with frequent pattern mining, the goal of PCA is to discover the statistically dominant patterns and thereby identify anomalies inside data. PCA can capture patterns in high-dimensional data by automatically choosing a (small) set of coordinates—the *principal components*—that reflect covariation among the original coordinates. Once we estimate these patterns from the archived and periodically updated data, we use them to transform the incoming data to make abnormal patterns easier to detect.

PCA detection also has a model estimation phase followed by an online detection phase. In the modeling phase, PCA captures the dominant pattern in a transformation matrix $\mathbf{PP}^T$, where $\mathbf{P}$ is formed by the top principal components chosen by PCA algorithm. Then in the online detection phase, the *abnormal component* of each message count vector $\mathbf{y}$ is computed as $\mathbf{y}_a = (\mathbf{I} - \mathbf{PP}^T)\mathbf{y}$, i.e., $\mathbf{y}_a$ is the projection of $\mathbf{y}$ onto the abnormal subspace. The *squared prediction error* $\mathbf{SPE} \equiv \|\mathbf{y}_a\|^2$ (squared length of vector $\mathbf{y}_a$) is used for detecting abnormal events: We mark vector $\mathbf{y}$ as abnormal if

$$\mathbf{SPE} = \|\mathbf{y}_a\|^2 > Q_\alpha, \qquad (5)$$

where Q_α denotes the threshold statistic for the $\mathbf{SPE}$ residual function at the $(1-\alpha)$ confidence level [11]. Due to limitations of space, we refer readers unfamiliar with these techniques to [28], [13] for details.

In a real deployment, the model can be updated periodically. Note that because of the noisier data in this phase and the workload-dependent nature of the non-pattern data, the model update period for PCA is usually shorter than that for frequent pattern mining.

VII. EVALUATION

We evaluate our approach with real logs from a 203-node Hadoop [1] installation on Amazon's EC2 cloud computing environment. Hadoop is an open source implementation of the MapReduce framework for large-scale parallel data processing. Hadoop is gaining popularity in both data mining and systems research, so it is important to understand its runtime behaviors, detect its execution anomalies and diagnose its performance degradation issues.

To compare our online approach directly against the offline algorithm proposed in [28], we replayed the same set of logs, containing over 24 million lines of log messages with an uncompressed size of 2.4GB. The logs were generated from 203 nodes running Hadoop for 48 hours, completing many standard MapReduce jobs such as distributed sort and text scan. The average machine load varies from fully utilized to mostly idle. The log contains 575,319 event traces, corresponding to 575,319 distinct file blocks in Hadoop File

#	Frequent sessions	Duration in sec (%ile)			Events
		99.90	99.95	99.99	
1	Allocated block, begin write	11	13	20	20.3%
2	Done write, update block map	7	8	14	44.6%
3	Delete block	-	-	-	12.5%
4	Serving block		—		3.8%
5	Read Exception (**see text**)		—		3.2%
6	Verify block		—		1.1%
	Total				**85.6%**

System (HDFS). We believe this dataset is representative of a production HDFS cluster.

In [28], all traces in the data set were labeled as normal or abnormal, together with the categories/explanations of most anomalies. This provides ground truth for evaluating our results. Over half a million event traces can be labeled because many traces are the same (and normal). Actually, there are only 680 distinct traces in the data. Notice that the labeling process does not take into account the durations of any traces. We show the effects of this omission later in this section.

To mimic how a system operator would use our technique, we evaluate our method with the following 2-step approach. First we randomly sample 10% of the execution traces, on which we construct the detection model, including the frequent patterns, the distributions of pattern durations, and the PCA detector. Then we replayed the entire trace and performed online problem detection using the derived model. This whole procedure is unsupervised, since we use the labels only for evaluation and not for building the model. We varied the subset of sampled data for building the model many times, and got identical detection results. This is mainly because the patterns we identify are so frequent that it is robust against random sampling.

There are two parameters that we need to set. The *maximum detection latency* T_{max} (defined in Section V) was set to 60 seconds, meaning the operator wants to be notified of a suspected anomaly at most 60 seconds after the suspect event trace appears in the log. The PCA threshold parameter α (described in Section VI). was set to 0.001, meaning that we are accepting less than 0.1% of all data points as abnormal (under certain assumptions [11]). These baseline values are chosen to be the common settings of such algorithms (or the most possible value to set without no understanding of the data), but in Section VII-B we show that our detection results are insensitive to these parameters over a wide range of values.

A. Stage 1 Pattern mining results

Recall that the goal of Stage 1 is to remove frequent patterns that presumably correspond to normal application behavior. Table I summarizes the frequent patterns found in the test data using our baseline parameter value $T_{max} = 60$s. Note first that the patterns identified encompass 85.6% of all events in the trace, so at most 14.4% of all events must be considered by Stage 2 (PCA anomaly detection).

The table shows, for example, that pattern 1 is the sequence of events corresponding to "Allocate a block for writing". 20.3% of the events in the trace are classified as belonging to an instance of this pattern and so will be filtered out and *not* passed to Stage 2. The duration of this pattern has a distribution whose 99.9, 99.95, and 99.99^{th} percentiles are 11, 13 and 20 seconds respectively. We choose these high percentile values because we want most normal sessions to complete within these intervals. Notice that even the 99.99th percentile of the pattern durations is significantly smaller than T_{max}; this is important since detection latency is based on the pattern duration or T_{max}, whichever is less. Due to space limitations, we only present detection results with T_{cut} set to the 99.95^{th} percentile values for each pattern. Results with other values are similar.

Patterns 1 and 2 are both related to writing a file block. They logically belong to the same operation, but a write session can be arbitrarily long: the application that writes the file may wait an arbitrary amount of time after the "begin write" before actually sending data. Since we are trying to keep detection latency below a finite threshold, we separate the beginning and ending sessions into two different patterns for timely detection. Obviously, there are certain limitations related to this separation, which we discuss in detail in Section VIII.

Patterns 4 to 6 contain only individual events. These events were used to report some numbers and do not contribute to event trace based detection, so a single event completes the operation (e.g., read, etc.).

Pattern 5 consists of an event that reports an exception, but as we discussed in [28], this is indeed *normal* operation and the message text represents a bad logging practice that has confused many users. In contrast, because we use pattern frequencies for detection, we easily recognize these exception messages as normal operation.

B. Detection precision and recall

We obtained the label/abnormal labels of each event trace from [28]. Since our technique is based on sessions (recall that a session is a subset of a trace), we determine a trace as abnormal if and only if it contains at least one abnormal session, allowing direct comparison using the original labels. We use the standard information-retrieval metrics of precision and recall to evaluate our approach. Let TP, FP, FN be the number of true positives, false positives, and false negatives, respectively. We have Precision = TP/(TP+FP) and Recall = TP/(TP+FN). 100% recall means that no actual problems were missed; 100% precision means there are no false alarms among those events identified as problems. In

Table II
DETECTION PRECISION AND RECALL.

(a) Varying α while holding $T_{max} = 60$

α	TP	FP	FN	Precision	Recall
0.0001	16,916	2,444	0	87.38%	100.00%
0.001	**16,916**	**2,748**	**0**	**86.03%**	**100.00%**
0.005	16,916	2,914	0	85.31%	100.00%
0.01	16,916	2,914	0	85.31%	100.00%

(b) Varying T_{max} while holding $\alpha = 0.001$

T_{max}	TP	FP	FN	Precision	Recall
15	2,870	129	14,046	95.70%	16.97%
30	16,916	2,748	0	86.03%	100.00%
60	**16,916**	**2,748**	**0**	**86.03%**	**100.00%**
120	16,916	2,748	0	86.03%	100.00%
240	14,233	2,232	2,683	86.44%	84.14%

our data set, there are 575,319 event traces of which 16,916 are labeled as anomalies.

Table II(a) varies the PCA confidence level α to show its effect on our precision and recall results, while Table II(b) varies the maximum detection delay T_{max}. The boldface rows of each table represent the baseline values $\alpha = 0.001$ and $T_{max} = 60$. The results show 100% recall over a wide range of values of α and T_{max}, meaning the algorithm captures every anomaly in the manual labels. The good recall is mainly due to strong patterns in the data: the event traces are direct representations of the program execution logic (which is likely to be deterministic and regular) as reflected by log printing statements. The strong patterns allows better tolerance to random noise, especially in the frequent pattern mining stage, where we can use a high support requirement to filter out random interleavings and reorderings.

The precision is not perfect due to false positives and some ambiguous cases. We review the false positives in detail when we compare with offline results in Section VII-D.

Furthermore, Table II(a) shows that precision and recall are largely insensitive to the choice of α over a wide range of values, consistent with the observations in both [28] and [13]. Table II(b) shows that precision and recall are insensitive to the maximum detection latency T_{max} over a certain range, but setting it outside this range (first and last rows of Table II(b)) adversely affects recall or precision. The intuition is that when T_{max} is too small, many logical sessions (especially those not covered by the dominant patterns) are cut off randomly, and when T_{max} is too large, many unrelated sessions are combined into the same message count vector, introducing too much noise for the PCA detector. Either effect degrades precision and recall.

As we described in Section V-B, we used a fairly sophisticated model to estimate the duration of sessions. If we had instead assumed a simple Gaussian distribution, the 99.95th percentile of T_{cut} would be estimated as 5.3s for Pattern 1 and 4.0s for Pattern 2 in Table I—less than half as long as the T_{cut} estimated by our distribution-fitting. Using the Gaussian-derived cutoff time, the number of false alarms increases by 45%, and precision falls to 80% from 86%. Therefore the small added complexity for duration
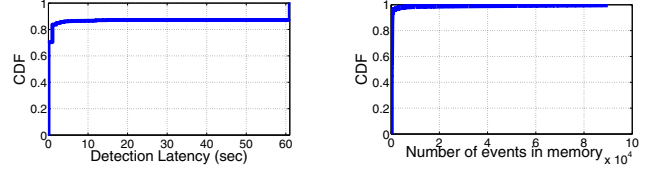

Figure 4. Detection latency and number of events kept in detector's buffer

distribution estimation (Section V-B) results in much better recall and precision.

C. Detection latency

Detection latency (defined in Section V) captures timeliness of detection, a key goal of our online approach. Recall that the difficulty of minimizing detection latency arises from the fact that it is not always possible to mark a trace "abnormal" until a specific event or set of events occurs. For example, the "allocate block" message in Pattern 1 of Table I simply indicates the start of a sequence of operations; the detector has to buffer the event and wait for further events. The final decision for this message is not reached until the last event of Pattern 1 (e.g., the last of the three expected "receiving" messages). Then the *detection time* for the trace containing this "allocate block" event is the time elapsed from the "allocate block" event being emitted to the time the detection result is made.

Figure 4 (Left) shows the cumulative distribution function (CDF) of detection times over all events. As expected, over 80% of the events can be determined as normal or abnormal within a couple of seconds. This is because we use the cutoff time T_{cut} to stop waiting for more events instead of the max latency T_{max} for most events. A few events require the maximum allowed detection latency: those that do not match any pattern (T_o defaults to ($T + T_{max}$)). By definition, these events are rare, so the overall impact of their longer detection time is limited.

Figure 4 (Right) shows the CDF of the number of events buffered in detector at every second. Because the detection time is low, most events are processed and removed from the buffer quickly. Thus as expected, the typical number of events in the buffer is small.

D. Comparison to offline results

Table III compares the offline detection results from [28] to our online detection results using baseline parameter values $\alpha = 0.001$ and $T_{max} = 60$. The error labels in the first column of the table were obtained directly from [28]. Not only do we successfully capture all anomalies as the offline method does, but we also get lower false negative rates. The reason is that for online detection, we segment an event trace into several sessions based on time duration, and base the detection on individual sessions rather than whole traces. Thus the data sent to the detector is free of noise resulting from application-dependent interleaving of multiple independent sessions (e.g., some blocks are read more often than others).

#	Anomaly Description	Actual	Offline	Online
1	Namenode not updated after deleting block	4297	4297	4297
2	Write exception client give up	3225	3225	3225
3	Write failed at beginning	2950	2950	2950
4	Replica immediately deleted	2809	2788	2809
5	Received block that does not belong to any file	1240	1228	1240
6	Redundant addStoredBlock	953	953	953
7	Delete a block that no longer exists on data node	724	650	724
8	Empty packet for block	476	476	476
9	Receive block exception	89	89	89
10	Replication monitor timedout	45	45	45
11	Other anomalies	108	107	108
	Total	**16916**	**16808**	**16916**

#	False Positive Description	Offline	Online
1	Normal background migration	1397	1403
2	Multiple replica (for task / job desc files)	349	368
	Total	**1746**	**1771**

#	Ambiguous Case	Offline	Online
	(see Section VII-D)	0	977

The two types of false positives in Table III are both "rare but normal events". For example, false positive #2 (over-replicating) is due to a special application request rather than a system problem. These are indeed rare events (only 368 occurrences across all traces) corresponding to rare but normal operations. These cases are hard to handle with a fully unsupervised detector. In order to handle these cases, we allow operators to manually add patterns to encode domain-specific knowledge about real problems and filter out these cases.

Table III lists ambiguous cases arising from the unclear definition of "anomaly". For example, our online algorithm marks some write sessions abnormal because one of the data nodes takes far longer to respond than all others do, resulting an unusually long writing session[1]. From the system administration point of view, these cases probably should be marked as anomalies, because although these blocks are eventually correctly written, this scenario effectively slows down the entire system to the speed of the slowest-responding node.

In [28], however, an event trace is labeled as normal if it contains all the events of a given pattern, without regard to *when* the events occur. Since they do not consider time information such as the durations of sessions, a scenario in which one data node takes a long time (but eventually

[1]The shortest duration for write sessions in this subset is 13 seconds, while the median duration for all sessions of this type is less than 1 second.

responds) is no different from a scenario in which all nodes respond in about the same amount of time. We consider session durations because we need to do so in order to bound the time to detection, but here we see that this additional information potentially improves the value of the online approach for operators in another way as well—by labeling as anomalous those event traces that are "correct but slow." If we consider slow operations problematic, at least some of these false alarms would instead be counted as a new type of anomaly not detected by the offline approach. To determine how many of the 977 ambiguous cases fall into this category, we would have to examine all event traces manually to evaluate duration lengths, in contrast to examining only distinct traces (without considering time information) as was done in [28]. However, we did an (informal) evaluation to estimate the number of cases that are probably due to this problem. We forced T_{cut} to 600 seconds for all patterns, which forces the detector to wait a long time for any incomplete patterns. Under these circumstances, the detection results approximate the results achieved by [28] when ignoring time information: the number of ambiguous cases drops to 314, which suggests at least 2/3 of these types of false alarms are fair to count as real anomalies. Nonetheless, to keep a fair comparison with the offline result, we stick to the original labels in all our evaluations.

VIII. Discussion

Limitations of online detection. An obvious limitation of online detection is that we cannot capture correlations across events over very long time periods. For example, as discussed in Section VII-A, there is a large and unpredictable time gap between Patterns 1 and 2 in Table I, so we must separate them into two patterns. However, a consequence of this separation is that we lose the ability to observe correlations between events in Pattern 1 and "matching" events in Pattern 2, which would potentially allow us to capture a new category of operational problems. For example, events in Pattern 1 indicate how many data nodes begin a write; each such node should have a corresponding "end write" event in Pattern 2.

This is an inherent limitation of online detection because of the detection latency requirement. This could be solved by remembering a longer history (maybe in a more compact/aggregated form), though that complicates the design of the detector. Thus we propose a different approach: by leveraging relatively cheap computing cycles, we can perform offline detection periodically on archived data to find anomalies violating such uncaptured constraints.

Use cases. In addition to showing individual anomaly alarms, our technique lets operators link each alarm back to the original logs and even the related source code segments, using the parsing and visualization techniques described in [28]. In addition, since we detect performance anomalies

quickly, operators have more time to prevent them from causing more serious errors. Anomalies due to deterministic bugs can recur frequently even over short timescales, as occurs with Anomaly 1 in Table III, which is due to a deterministic bug in the Hadoop source code. Since alarming on each occurrence would overwhelm the operator's attention, we cluster the anomalies hierarchically and report the count of each anomaly *type*. Space limitations prevent a description of our clustering method.

IX. Conclusions and Future Work

We showed how to use a two-stage data mining technique to identify and filter out common (normal) operational patterns from free-text console logs, and then perform PCA-based anomaly detection on the remaining patterns to identify operational problems within minutes of their occurrence (as represented by information in the console logs). Our approach, validated on real data, addresses a key need for operators of such large systems, and matches or outperforms current offline methods for free-text log analysis [28] that could not be used in an online setting.

As future work, we plan to monitor console logs from multiple components of the system (e.g. both the file system and the application that uses it), and automatically determine which component is responsible when a particular problem is detected.

X. Acknowledgments

References

[1] D. Borthakur. The hadoop distributed file system: Architecture and design. Hadoop Project Website, 2007.

[2] M. Y. Chen and et al. Pinpoint: Problem determination in large, dynamic internet services. In *Proc. IEEE DSN '02*, Washington, DC, 2002.

[3] M. Y. Chen and et al. Path-based failure and evolution management. In *Proc. NSDI'04*, San Francisco, CA, 2004.

[4] A. Clauset, C. Shalizi, and M. Newman. Power-law distributions in empirical data. *SIAM Review*, 2009.

[5] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet. In *Proceedings of SIGCOMM*, 1999.

[6] R. Fonseca and et al. Xtrace: A pervasive network tracing framework. In *In Proc. NSDI*, 2007.

[7] J. Han, H. Cheng, D. Xin, and XifengYan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1), 2007.

[8] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with Swatch. In *Proc. USENIX LISA '93*, pages 145–152, 1993.

[9] J. Hellerstein, S. Ma, and C. Perng. Discovering actionable patterns in event data. *IBM Sys. Jour*, 41(3), 2002.

[10] J. R. I.M. Chakravarti, R.G. Laha. *Handbook of Methods of Applied Statistic*, volume I. John Wiley and Sons, 1967.

[11] J. E. Jackson and G. S. Mudholkar. Control procedures for residuals associated with principal component analysis. *Technometrics*, 21(3):341–349, 1979.

[12] T. Karagiannis, K. Papagiannaki, N. Taft, and M. Faloutsos. Profiling the end hos. In *Proceedings of Passive and Active Measurement Workshop (PAM)*, Belgium, 2007.

[13] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proc. ACM SIGCOMM*, 2004.

[14] C. LaRosa and et al. Frequent pattern mining for kernel trace data. In *Proc. of ACM SAC'08*, 2008.

[15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings OSDI'04*, 2004.

[16] P. Louridas, D. Spinellis, and V. Vlachos. Power laws in software. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 18, 2008.

[17] S. Ma and J. L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *Proc. IEEE ICDE*, Washington, DC, 2001.

[18] OSSEC.org. *OSSEC Manual*, 2008.

[19] J. Pei and et al. PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceeding of IEEE ICDE'01*, Heidelberg, Germany.

[20] J. E. Prewett. Analyzing cluster log files using logsurfer. In *Proc. Annual Conf. on Linux Clusters*, 2003.

[21] K. Rieck and et al. Computation of similarity measures for sequential data using generalized suffix trees. In *NIPS'2007*. MIT Press, Cambridge, MA, 2007.

[22] R. Srikant and R. Agrawa. Mining sequential patterns: generalizations and performance improvements. In *Proceeding of EDBT'96*, Avignon, France, 1996.

[23] J. Stearley. Towards informatic analysis of syslogs. In *Proc. IEEE CLUSTER*, Washington, DC, 2004.

[24] R. Vaarandi. A data clustering algorithm for mining patterns from event logs. *Proc. IPOM*, 2003.

[25] R. Vaarandi. A breadth-first algorithm for mining frequent patterns from event logs. In *INTELLCOMM*, volume 3283, pages 293–308. Springer, 2004.

[26] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *Proceedings of OSDI'04*, 2004.

[27] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling internet backbonetraffic: Behavior models and applications. In *Proceedings of ACM SIGCOMM*, 2005.

[28] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Large-scale system problems detection by mining console logs. In *Proceedings of SOSP'09*, Oct. 2009.

[29] K. Yamanishi and et al. Dynamic syslog mining for network failure monitoring. In *Proc. KDD'05*, New York, NY, 2005.

[30] Y. Ye and et al. IMDS: Intelligent malware detection system. In *Proceedings of ACM SIGKDD'07*, 2007.

[31] M. J. Zaki. Spade: an efficient algorithm for mining frequent sequences. *Machine Learning*, 42:31–60, 2004.

[32] A. Zheng and et al. Statistical debugging: Simultaneous isolation of multiple bugs. In *Proceedings of ICML'06*, 2006.