Detecting Large-Scale System Problems by Mining Console Logs

Author : Wei Xu, Ling Huang, Armando Fox,
David Patterson, Michael Jordan

Conference: ICML 2010
SOSP2009, ICDM 2009

Reporter: Zhe Fu

Outline

SOSP 09


Detecting Large-Scale System Problems by Mining Console Logs

Offline Analysis

ICDM 09


Online System Problem Detection by Mining Patterns of Console Logs

Online Detection

ICML 10

Invited Applications Paper


Detecting Large-Scale System Problems by Mining Console Logs

Invited Applications Paper

Outline

- Introduction
- Key Insights
- Methodology
- Evaluation
- Online Detection
- Conclusion

Introduction

Background of console logs

- Console logs rarely help in large-scale datacenter services
- Logs are too large to examine manually and too unstructured to analyze automatically
- It’s difficult to write rules that pick out the most relevant sets of events for problem detection

Anomaly detection

- Unusual log messages often indicate the source of the problem

Introduction

Related work:

- as a collection of English words
- as a single sequence of repeating events

Contributions:

- A general methodology for automated console log processing
- Online problem detection with message sequences
- System implementation and evaluation on real world systems.

Key Insights

Insight 1: Source code is the "schema" of logs.

- Logs are quite structured because generated entirely from a relatively small set of log printing statements in source code.

```
starting: xact 325 is COMMITTING
starting: xact 346 is ABORTING
```

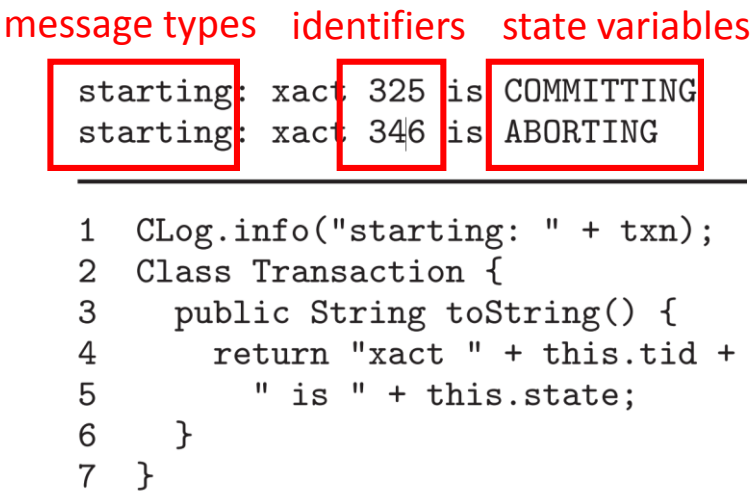
```
1  CLog.info("starting: " + txn);
2  Class Transaction {
3    public String toString() {
4      return "xact " + this.tid +
5        " is " + this.state;
6    }
7  }
```

- Our approach can accurately parse all possible log messages, even the ones rarely seen in actual logs.

Key Insights

Insight 2: Common log structures lead to useful features.

- Message types: marked by constant strings in a log message
- Message variables:
  - Identifiers: variables that identify an object manipulated by the program
  - state variables: labels that enumerate a set of possible states an object could have in program



Key Insights

Insight 2: Common log structures lead to useful features.

Table 1. State variables and identifiers

Variable	**Examples**	**Distinct values**
Identifiers	`transaction_id` in Darkstar; `block_id` in Hadoop FS; `cache_key` in Apache server; `task_id` in map-reduce.	many
State Vars	Transaction stages in Darkstar; Server names in Hadoop; HTTP status code (200, 404); POSIX process return values.	few

Key Insights

Insight 3: Message sequences are important in problem detection.

- Messages containing a certain file name are likely to be highly correlated because they are likely to come from logically related execution steps in the program.
- Many anomalies are only indicated by incomplete message sequences.

For example, if a write operation to a file fails silently (perhaps because the developers do not handle the error correctly), no single error message is likely to indicate the failure.

Key Insights

Insight 4: Logs contain strong patterns with lots of noise.

- normal patterns—whether in terms of frequent individual messages or frequent message sequences—are very obvious

  frequent pattern mining and Principal Component Analysis (PCA)

- Two most notable kinds of noise
  - random interleaving of messages from multiple threads or processes
  - inaccuracy of message ordering)

  grouping methods

Case Study

System	Lang	Logger	Msg Construction	Lines of Code	Lines of Logs	Vars	Parse	ID	ST
Operating system									
Linux (Ubuntu)	C	custom	printk + printf wrap	7477k	70817	70506	Y	Y[b]	Y
Low level Linux services									
Bootp	C	custom	printf wrap	11k	322	220	Y	N	N
DHCP server	C	custom	printf wrap	23k	540	491	Y	Y[b]	Y
DHCP client	C	custom	printf wrap	5k	239	205	Y	Y[b]	Y
ftpd	C	custom	printf wrap	3k	66	67	Y	Y	N
openssh	C	custom	printf wrap	124k	3341	3290	Y	Y	Y
crond	C	printf	printf wrap	7k	112	131	Y	N	Y
Kerboros 5	C	custom	printf wrap	44k	6261	4971	Y	Y	Y
iptables	C	custom	printf wrap	52k	2341	1528	Y	N	Y
Samba 3	C	custom	printf wrap	566k	8461	6843	Y	Y	Y
Internet service building blocks									
Apache 2	C	custom	printf wrap	312k	4008	2835	Y	Y	Y
mysql	C	custom	printf wrap	714k	5092	5656	Y	Y[b]	Y[b]
postgresql	C	custom	printf wrap	740k	12389	7135	Y	Y[b]	Y[b]
Squid	C	custom	printf wrap	148k	2675	2740	Y	Y	Y
Jetty	Java	log4j	string concatenation	138k	699	667	Y	Y	Y
Lucene	Java	custom	custom log function	217k	143	159	Y[a]	Y	N
BDB (Java)	Java	custom	custom trace	260k	-	-	-	Y	N
Web Applications									
MoinMoin	Python	custom	string replacement	96k	566	611	Y	Y	Y
Trac	Python	custom	string replacement	84k	104	65	Y	Y	Y
AppEngine SDK	Python	custom	string replacement	122k	378	349	Y	Y	Y
Flash Game Client	ActionScript	AS3 built-in	string concatenation	-[d]	-[d]	-[d]	Y	Y	Y
Distributed systems									
Hadoop	Java	custom log4j	string concatenation	173k	911	1300	Y	Y	Y
Darkstar	Java	jdk-log	Java format string	90k	578	658	Y	Y[b]	Y[b]
Nutch	Java	log4j	string concatenation	64k	507	504	Y	Y	Y
Cassandra	Java	log4j	string concatenation	46k	393	437	Y	N	Y
Storage Prototype	C	custom	custom trace	-[c]	-[c]	-[c]	-[c]	Y	Y
Google System 1	C++	custom	string concatenation	-[d]	10k	-[d]	Y	Y	Y
Google System 2	C++	custom	string concatenation	-[d]	21k	-[d]	Y	Y	Y
Google System 3	C++	custom	string concatenation	-[d]	6k	-[d]	Y	Y	Y

Methodology

Step 1: Log parsing

- Convert a log message from unstructured text to a data structure

Step 2: Feature creation

- Constructing the *state ratio vector* and the *message count vector* features

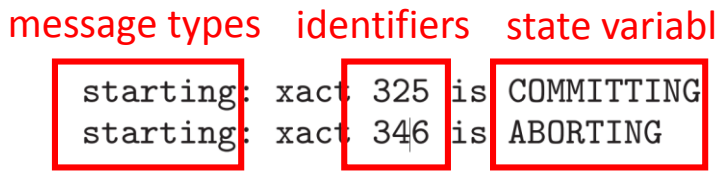
Step 3: Machine learning

- Principal Component Analysis(PCA)-based anomaly detection method

Step 4: Visualization

- Decision tree

Step 1: Log parsing



regular expression:
`starting: xact (. *) is (. *)`

- Challenge: Templatize automatically
  - C language
    - `fprintf(LOG, "starting: xact %d is %s")`
  - Java
    - `CLog.info("starting: " + txn)`
- Difficulty in OO (object-oriented) language
  - We need to know that `CLog` identifies a logger object
  - OO idiom for printing is for an object to implement a `toString()` method that returns a printable representation of itself for interpolation into a string
  - Actual `toString()` method used in a particular call might be defined in a subclass rather than the base class of the logger object

Step 1: Log parsing

Parsing Approach - Source Code

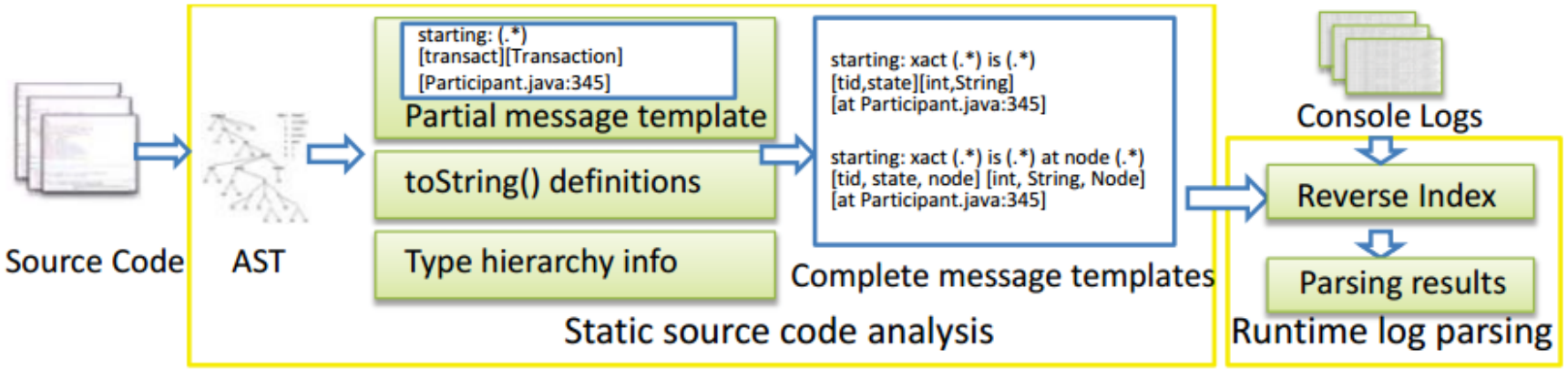
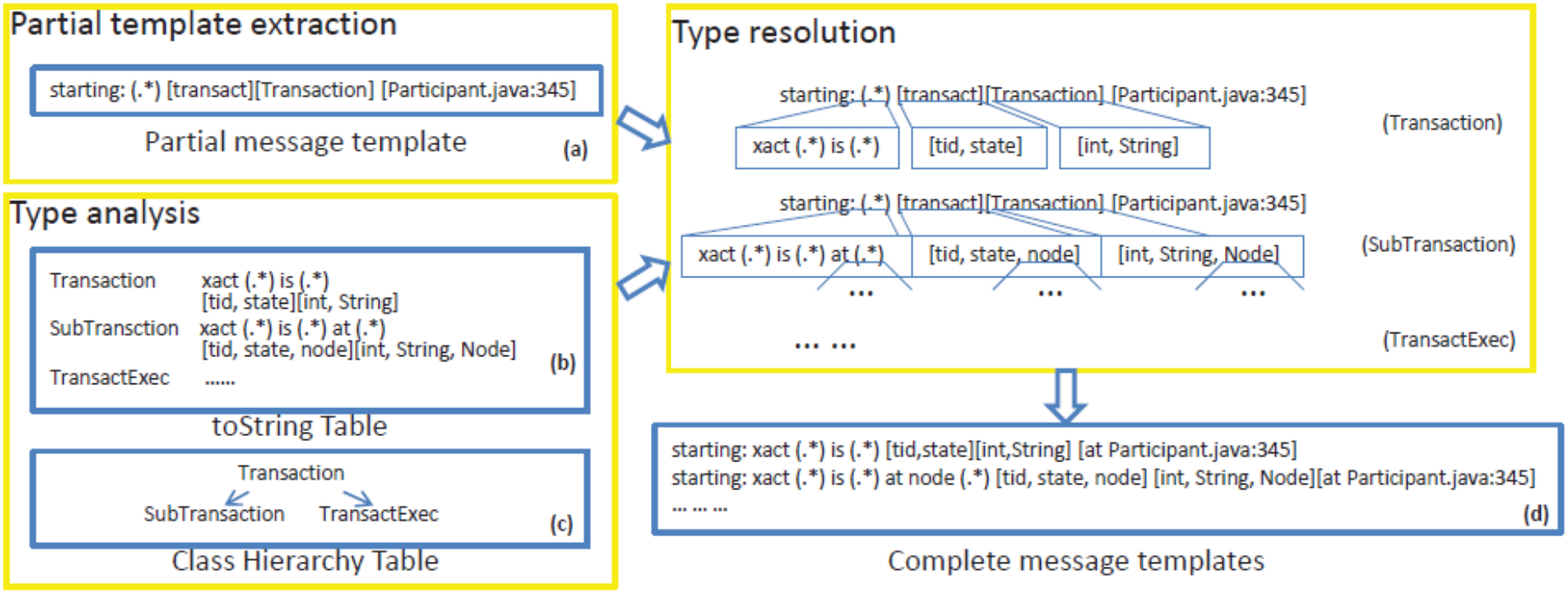


Figure 3: Using source code information to parse console logs.

Step 1: Log parsing

Parsing Approach - Source Code



Step 1: Log parsing

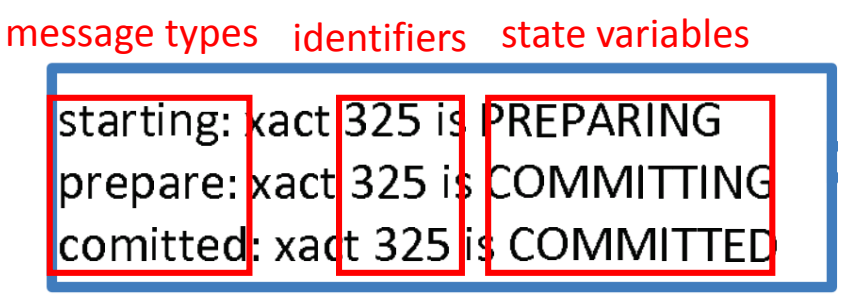
Parsing Approach - Logs

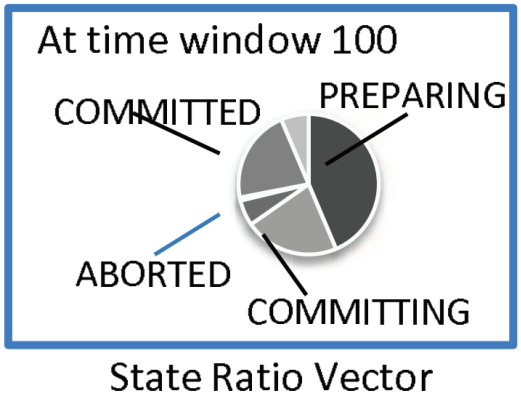
- Apache Lucene reverse index
- Implement as a Hadoop map-reduce job
  - Replicating the index to every node and partitioning
  - The map stage performs the reverse-index search
  - The reduce stage processing depends on the features to be constructed

Step 2: Feature creation

State ratio vector

- Each state ratio vector: a group of *state variables* in a time window
- Each vector dimension: a distinct state variable value
- Value of the dimension: how many times this state value appears in the time window





State Ratio Vector

choose state variables that were reported at least $0.2N$ times

choose a size that allows the variable to appear at least $10D$ times in 80% of all the time windows

Step 2: Feature creation

Message count vector

- Each message count vector: group together messages with the same identifier values
- Each vector dimension: different message type
- Value of the dimension: how many messages of that type appear in the message group

message types identifiers state variables

starting: xact 325 is PREPARING
prepare: xact 325 is COMMITTING
comitted: xact 325 is COMMITTED

325: 1 1 1 0 0 0 0 0 0
326: 1 0 1 0 0 0 0 0 0
327: 1 1 1 0 1 0 0 0 0

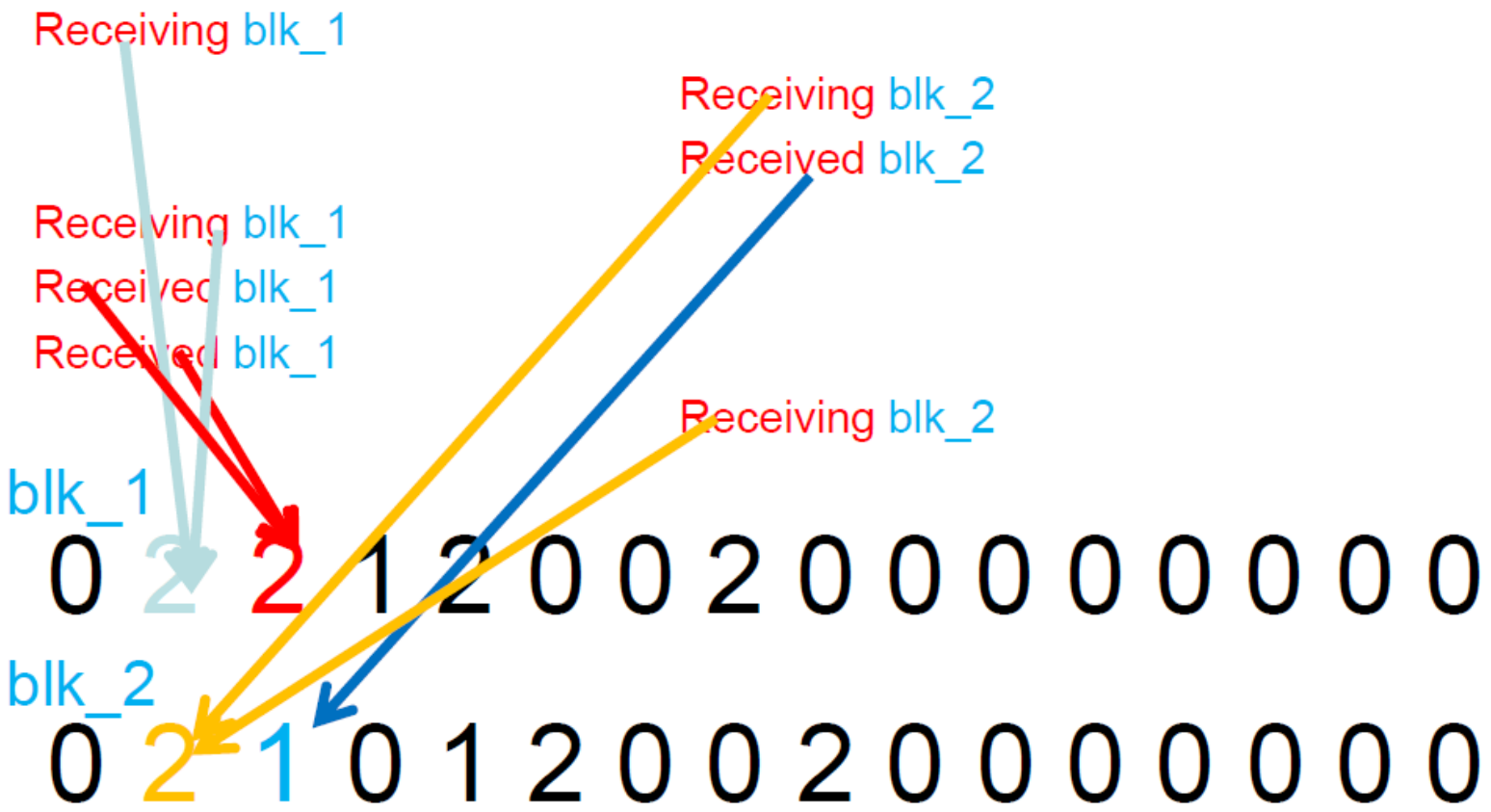
Message Count Vectors

Algorithm 1 Message count vector construction

1. Find all message variables reported in the log with the following properties:
   a. Reported many times;
   b. Has many distinct values;
   c. Appears in multiple message types.
2. Group messages by values of the variables chosen above.
3. For each message group, create a message count vector $\mathbf{y} = [y_1, y_2, \ldots, y_n]$, where y_i is the number of appearances of messages of type i $(i = 1 \ldots n)$ in the message group.

Step 2: Feature creation

Message count vector



Step 2: Feature creation

State ratio vector

- Capture the aggregated behavior of the system over a time window

Message count vector

- help detect problems related to individual operations

Feature	Rows	Columns
Status ratio matrix $\mathbf{Y}^s$	time window	state value
Message count matrix $\mathbf{Y}^m$	identifier	message type

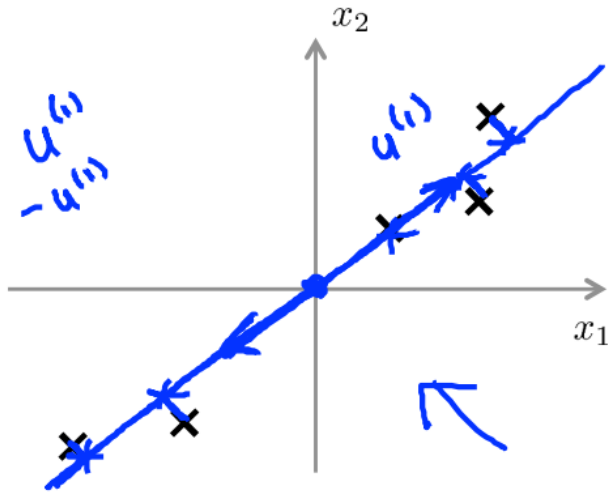
Table 4: Semantics of rows and columns of features

Also implement as a Hadoop map-reduce job

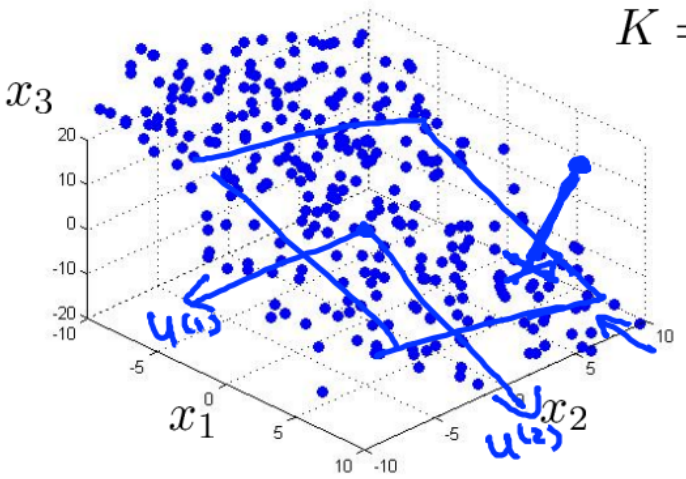
Step 3: Machine learning

Principal Component Analysis (PCA)-based anomaly detection

Principal Component Analysis (PCA) problem formulation



$3D \rightarrow 2D$
$K = 2$



Reduce from 2-dimension to 1-dimension: Find a direction (a vector $u^{(1)} \in \mathbb{R}^n$) onto which to project the data so as to minimize the projection error.

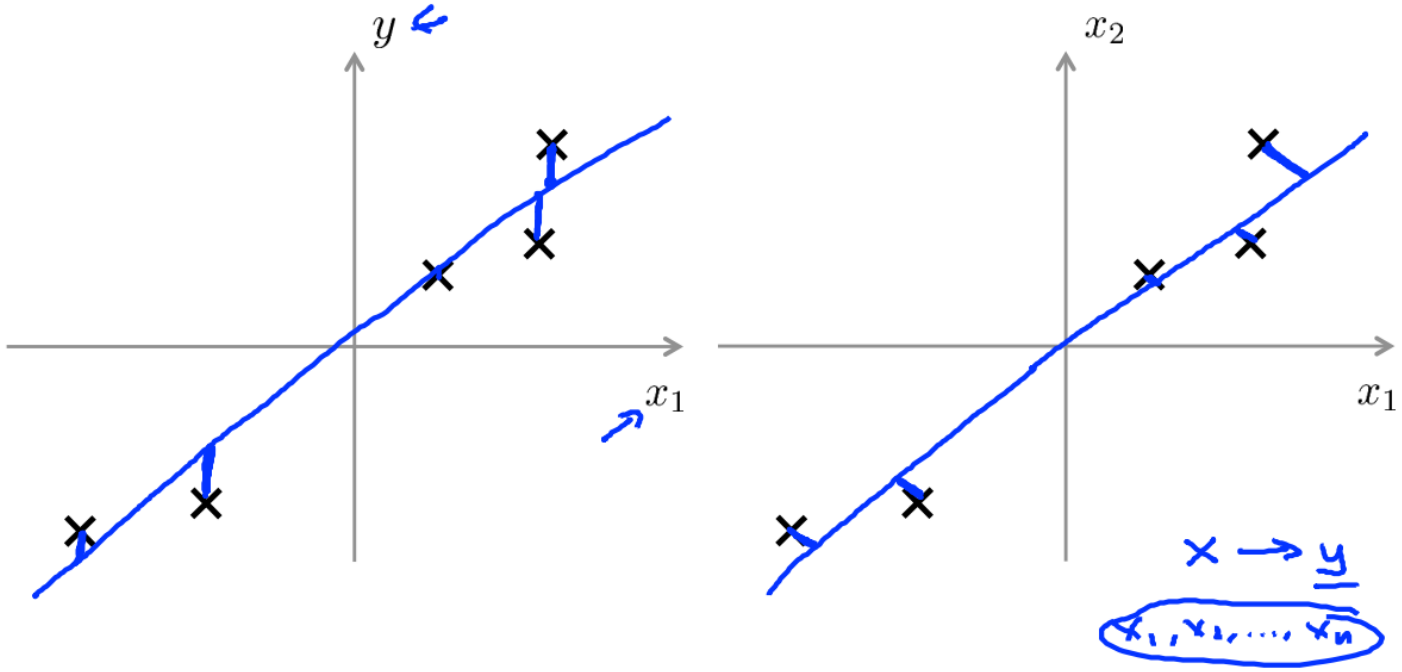
Reduce from n-dimension to k-dimension: Find k vectors $u^{(1)}, u^{(2)}, \ldots, u^{(k)}$ onto which to project the data, so as to minimize the projection error.

Andrew Ng

Step 3: Machine learning

Principal Component Analysis (PCA)-based anomaly detection

PCA is not linear regression



Andrew Ng

Step 3: Machine learning

Intuition behind PCA anomaly detection

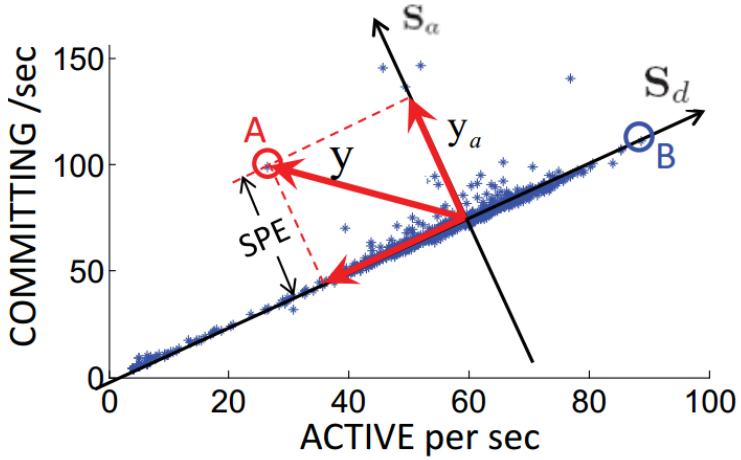


Figure 4: The intuition behind PCA detection with simplified data. We plot only two dimensions from the Darkstar state variable feature. It is easy to see high correlation between these two dimensions. PCA determines the dominant normal pattern, separates it out, and makes it easier to identify anomalies.

Feature data sets	n	k
Darkstar - message count	18	3
Darkstar - state ratio	6	1
HDFS - message count	28	4
HDFS - state ratio	202	2

Table 5: Low effective dimensionality of feature data. n = Dimensionality of feature vector $\mathbf{y}$; k = Dimensionality required to capture 95% of variance in the data. In all of our data, we have $k \ll n$, exhibiting low effective dimensionality.

Step 3: Machine learning

Principal Component Analysis (PCA)-based anomaly detection

Data preprocessing

Training set: $x^{(1)}, x^{(2)}, \ldots, x^{(m)}$

Preprocessing (feature scaling/mean normalization):

$$\mu_j = \frac{1}{m} \sum_{i=1}^{m} x_j^{(i)}$$

Replace each $x_j^{(i)}$ with $x_j - \mu_j$.

If different features on different scales (e.g., $x_1 =$ size of house, $x_2 =$ number of bedrooms), scale features to have comparable range of values.

$$x_j^{(i)} \leftarrow \frac{x_j^{(i)} - \mu_j}{s_j}$$

Andrew Ng

Step 3: Machine learning

Principal Component Analysis (PCA)-based anomaly detection

Principal Component Analysis (PCA) algorithm

Reduce data from n-dimensions to k-dimensions

Compute "covariance matrix":

$$\Sigma = \frac{1}{m} \sum_{i=1}^{n} (x^{(i)})(x^{(i)})^T$$

$n \times 1$ $1 \times n$ $n \times n$

Sigma

Compute "eigenvectors" of matrix Σ:

```
[U,S,V] = svd(Sigma);
```

→ Singular value decomposition

eig(Sigma)

$n \times n$ matrix

$$U = \begin{bmatrix} | & | & | & & | \\ u^{(1)} & u^{(2)} & u^{(3)} & \cdots & u^{(n)} \\ | & | & | & & | \end{bmatrix}$$

k

$U \in \mathbb{R}^{n \times n}$

$u^{(1)}, \ldots, u^{(k)}$

Andrew Ng

Step 3: Machine learning

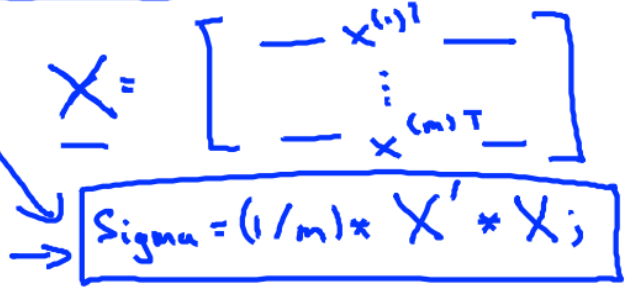
Principal Component Analysis (PCA)-based anomaly detection

Principal Component Analysis (PCA) algorithm summary

- After mean normalization (ensure every feature has zero mean) and optionally feature scaling:

$$\text{Sigma} = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})(x^{(i)})^T$$

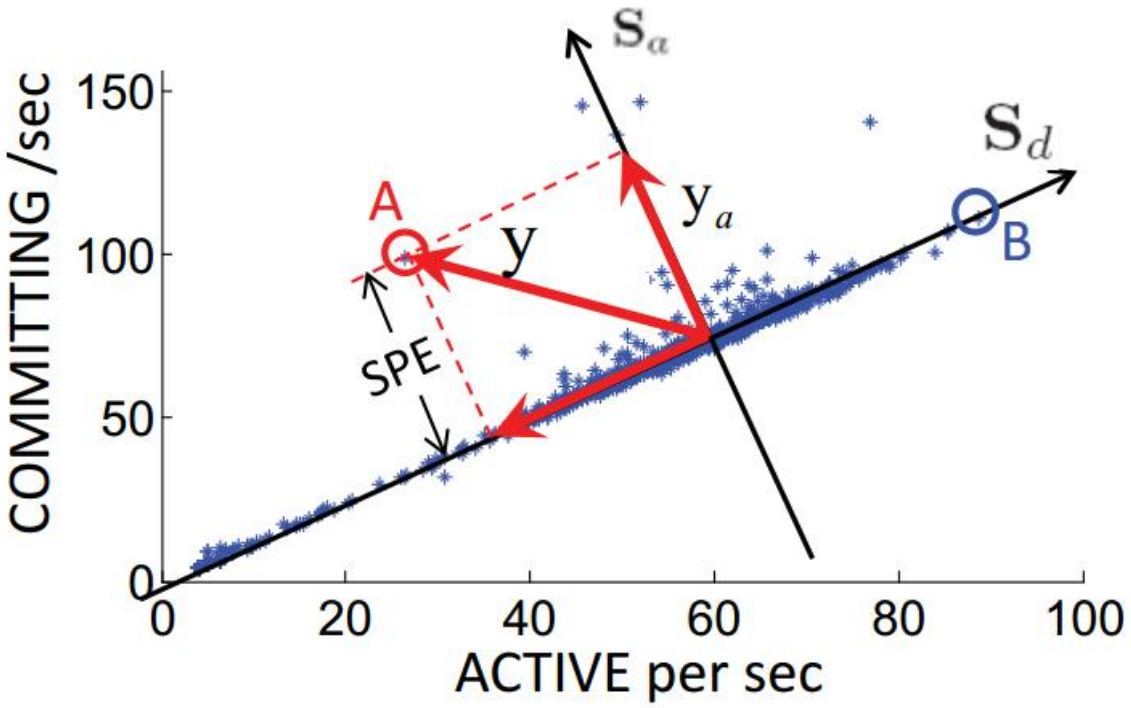
```
[U,S,V] = svd(Sigma);
Ureduce = U(:,1:k);
z = Ureduce'*x;
```



$x \in \mathbb{R}^n$ $x_0 = 1$

Step 3: Machine learning

Principal Component Analysis (PCA)-based anomaly detection



$$\mathbf{y}_a = (\mathbf{I} - \mathbf{P}\mathbf{P}^T)\mathbf{y}$$

$$\mathbf{P} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k]$$

$$\mathbf{SPE} = \|\mathbf{y}_a\|^2 > Q_\alpha$$

Step 3: Machine learning

Improving PCA detection results

- Applied Term Frequency / Inverse Document Frequency (TF-IDF)

$$w_{i,j} \equiv y_{i,j} \log(n/df_j)$$

where df_j is total number of message groups that contain the j-th message type

- Using better similarity metrics and data normalization

$$\mathcal{K}(\mathbf{x}, \mathbf{y}) = \frac{\mathbf{x} \cdot \mathbf{y}}{\sqrt{\mathbf{x} \cdot \mathbf{x}}\sqrt{\mathbf{y} \cdot \mathbf{y}}}$$

Step 4: Visualization

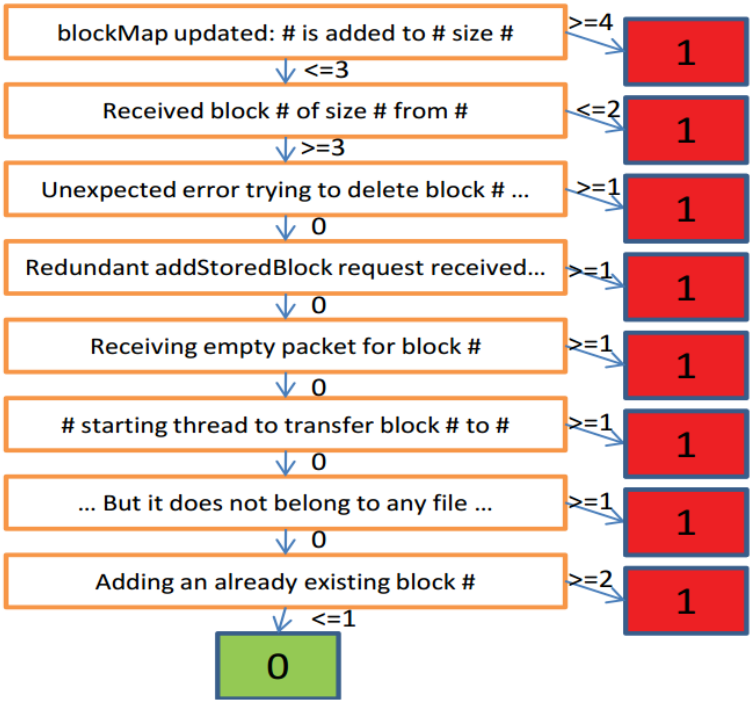
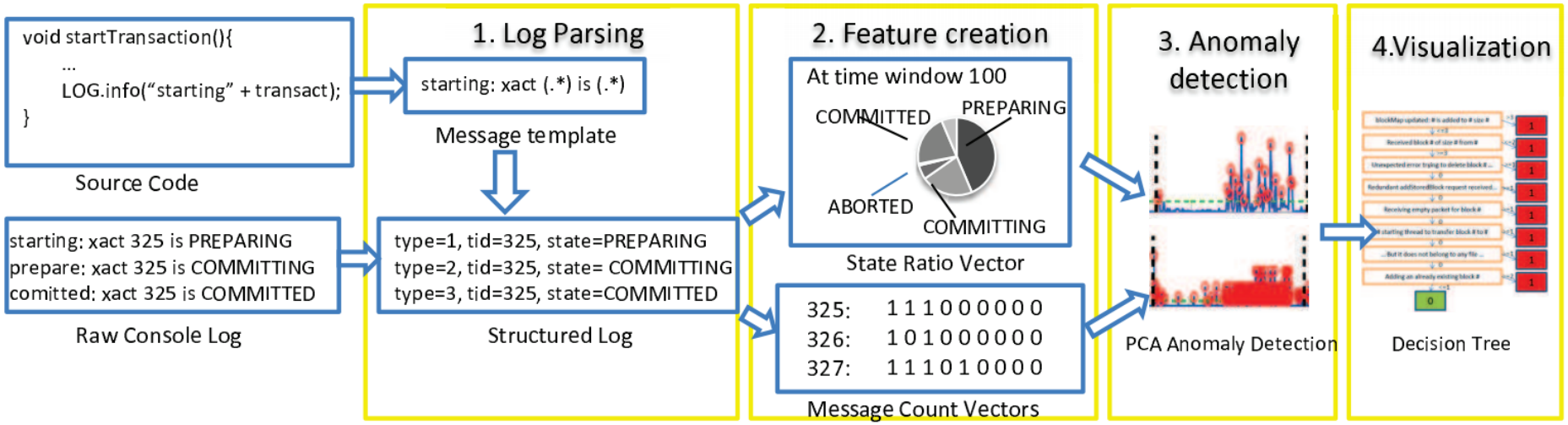


Figure 7: The decision tree visualization. Each node is the message type string (# is the place holder for variables). The number on the edge is the threshold of message count, generated by the decision tree algorithm. Small boxes contain the labels from PCA, with a red 1 for abnormal and a green 0 for normal.

Methodology



Evaluation

Dataset:

Table 2. Data sets used in evaluation. Nodes=Number of nodes in the experiments.

System	Nodes	Messages	Log Size
Darkstar	1	1,640,985	266 MB
Hadoop (HDFS)	203	24,396,061	2412 MB

- From Elastic Compute Cloud (EC2)
- 203 nodes of HDFS and 1 nodes of Darkstar

Evaluation

Parsing accuracy:

System	Total Log	Failed	Failed %
HDFS	24,396,061	29,636	0.121%
Darkstar	1,640,985	35	0.002%

Table 6: Parsing accuracy. Parse fails on a message when we cannot find a message template that matches the message and extract message variables.

Parse fails when cannot find a message template that matches the message and extract message variables.

Evaluation

Scalability:

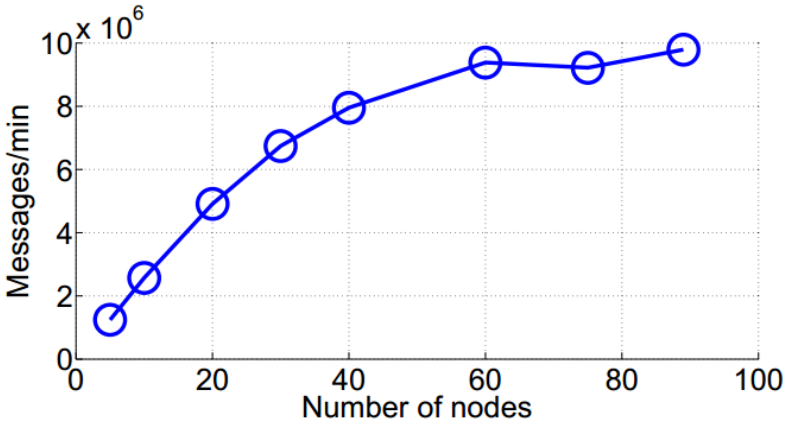


Figure 5: Scalability of log parsing with number of nodes used. The x-axis is the number of nodes used, while the y-axis is the number of messages processed per minute. All nodes are Amazon EC2 *high-CPU medium* instances. We used the HDFS data set (described in (Table 3) with over 24 million lines. We parsed raw textual logs and generated the message count vector feature (see Section 4.2). Each experiment was repeated 4 times and the reported data point is the mean.

50 nodes, takes less than 3 minutes , less than 10 minutes with 10 node

Evaluation

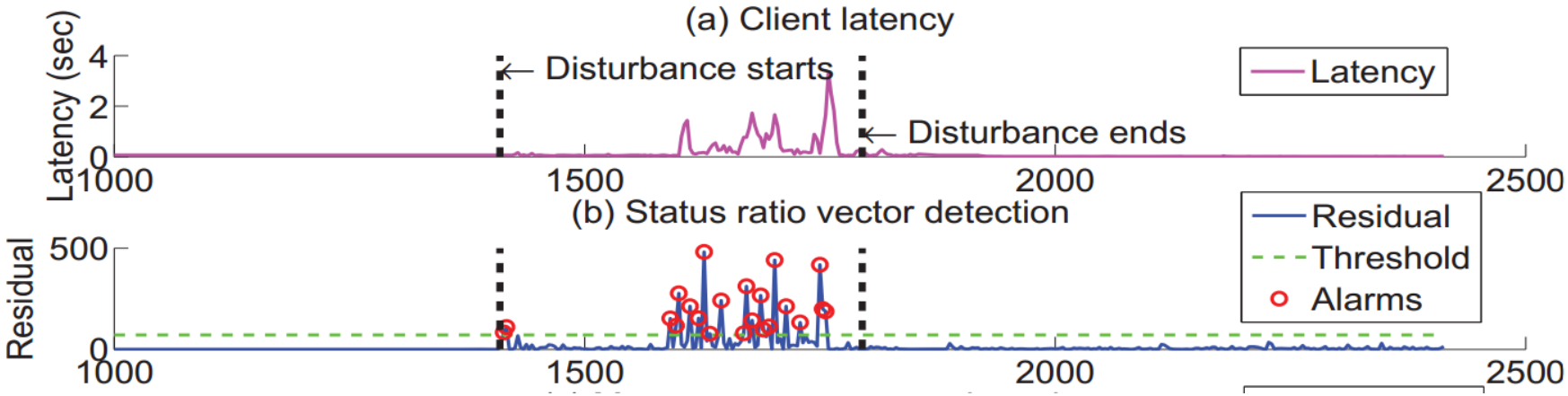
Darkstar

- DarkMud Provided by the Darkstar team
- Emulate 60 user clients in the DarkMud virtual world performing random operations
- Run the experiment for 4800 seconds
- Injected a performance disturbance by capping the CPU available to Darkstar to 50% during time 1400 to 1800 sec

Evaluation

Darkstar - state ratio vectors

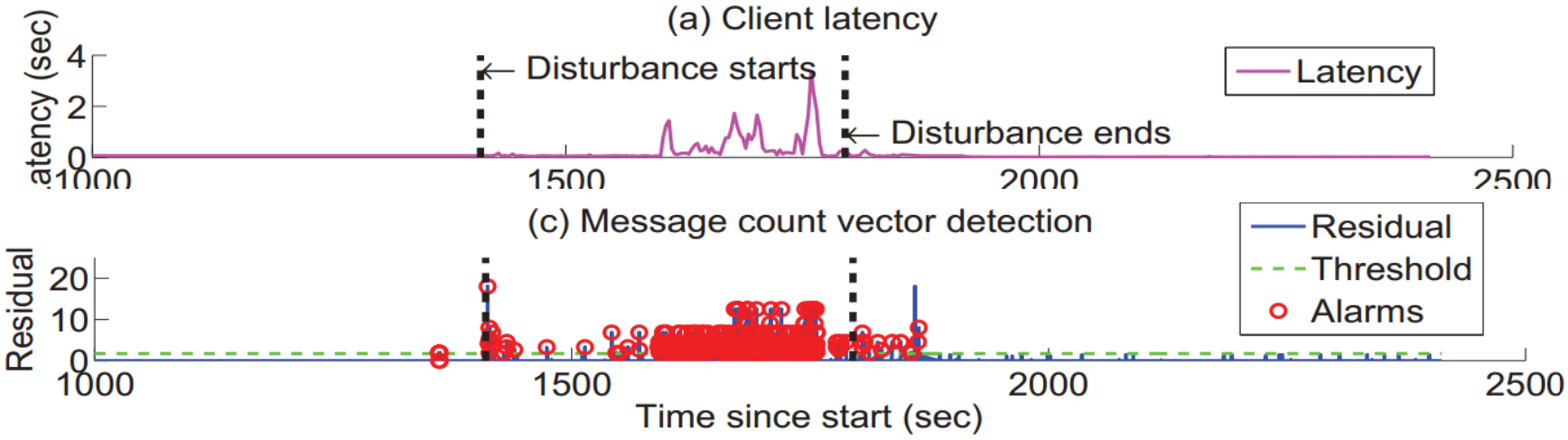
- 8 distinct values, including `PREPARING`, `ACTIVE`, `COMMITTING`, `ABORTING` and so on
- Ratio between number of `ABORTING` to `COMMITTING` increases from about 1:2000 to about 1:2
- Darkstar does not adjust transaction timeout accordingly



Evaluation

Darkstar - message count vectors

- 68,029 transaction ids reported in 18 different message types, Y^m is 68,029×18
- PCA identifies the normal vectors: {`create`, `join txn`, `commit`, `prepareAndCommit`}
- Augmented each feature vector using the timestamp of the last message in that group



Evaluation

Hadoop

- Set up a Hadoop cluster on 203 EC2 nodes
- Run sample Hadoop map-reduce jobs for 48 hours
- Generate and processing over 200 TB of random datas
- Collect over 24 million lines of logs from HDFS

Evaluation

Hadoop - message count vectors

- Automatically chooses one identifier variable, the `blockid`, which is reported in 11,197,954 messages (about 50% of all messages) in 29 message types.
- Y^m has a dimension of 575, 139×29

Evaluation

Hadoop - message count vectors

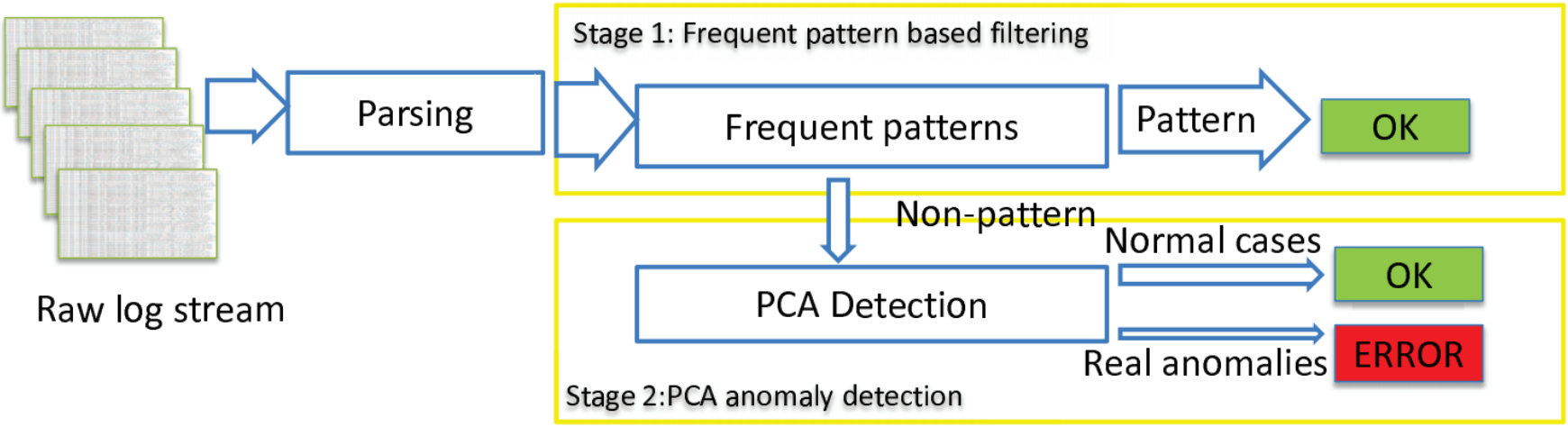
#	Anomaly Description	Actual	Raw	TF-IDF
1	Namenode not updated after deleting block	4297	475	4297
2	Write exception client give up	3225	3225	3225
3	Write failed at beginning	2950	2950	2950
4	Replica immediately deleted	2809	2803	2788
5	Received block that does not belong to any file	1240	20	1228
6	Redundant addStoredBlock	953	33	953
7	Delete a block that no longer exists on data node	724	18	650
8	Empty packet for block	476	476	476
9	Receive block exception	89	89	89
10	Replication Monitor timedout	45	37	45
11	Other anomalies	108	91	107
	Total	**16916**	**10217**	**16808**

#	False Positive Description	Raw	TF-IDF
1	Normal background migration	1399	1397
2	Multiple replica (for task / job desc files)	372	349
3	Unknown Reason	26	0
	Total	**1797**	**1746**

Table 7: Detected anomalies and false positives using PCA on Hadoop message count vector feature. Actual is the number of anomalies labeled manually. Raw is PCA detection result on raw data, TF-IDF is detection result on data preprocessed with TF-IDF and normalized by vector length (Section 5).

- The first anomaly in Table 7 uncovered a bug that has been hidden in HDFS for a long time.
  - no single error message indicating the problem
- we do not have the problem that causes confusion in traditional grep based log analysis.
  - `#:Got Exception while serving # to #:#`
- Algorithm does report some false positives, which are inevitable
  - a few blocks are replicated 10 times instead of 3 times for the majority of blocks.

Online Detection



Two Stage Online Detection Systems

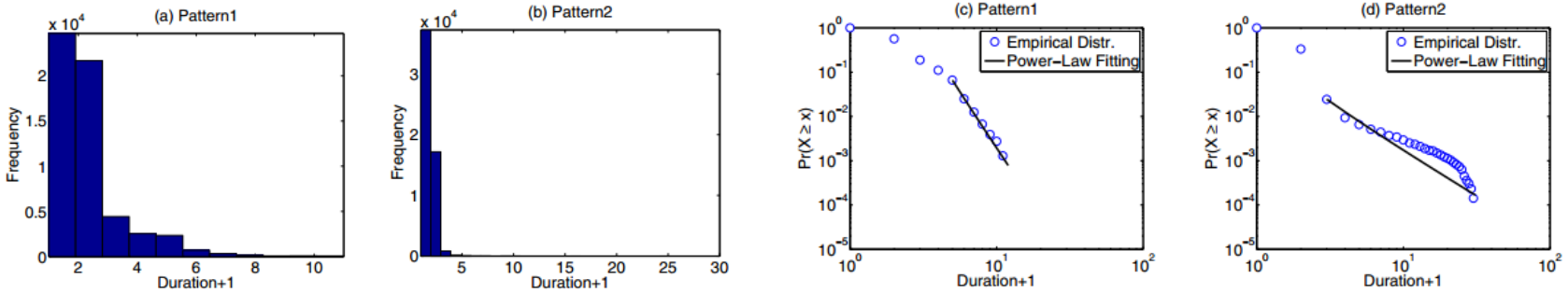
Stage 1: Frequent pattern based filtering

- *event trace*: a group of events that reports the same identifier.
- *session*: a subset of closely-related events in the same event trace that has a predictable duration.
- *duration*: the time difference between the earliest and latest timestamps of events in the session.
- *frequent pattern:* a session with its duration distribution:
  - 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}
- T_{max}: a user-specified maximum allowable *detection latency*
- *detection latency:* the time between an event occurring and the decision of whether the event is normal or abnormal

Stage 1: Frequent pattern based filtering

- *A. Combining time and sequence information*
  - *Step1*: Use time gaps to find first session in each execution trace (coarsely)

    the time gap size is a configurable parameter
  - *Step2*: Identify the dominant session
  - *Step3*: Refine result using the frequent session and compute duration statistics
- *B. Estimating distributions of session durations*



power-law distribution

Online Detection - Evaluation

A. Stage 1 Pattern mining results

Table I

FREQUENT PATTERNS MINED. PATTERN 3'S DURATION CANNOT BE ESTIMATED BECAUSE THE DURATIONS ARE TOO SMALL TO CAPTURE IN TRAINING SET. PATTERNS 4–6 CONSIST OF ONLY A SINGLE EVENT EACH AND THUS HAVE NO DURATIONS.

#	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%**

Online Detection - Evaluation

B. Detection precision and recall

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%

Online Detection - Evaluation

C. Detection latency

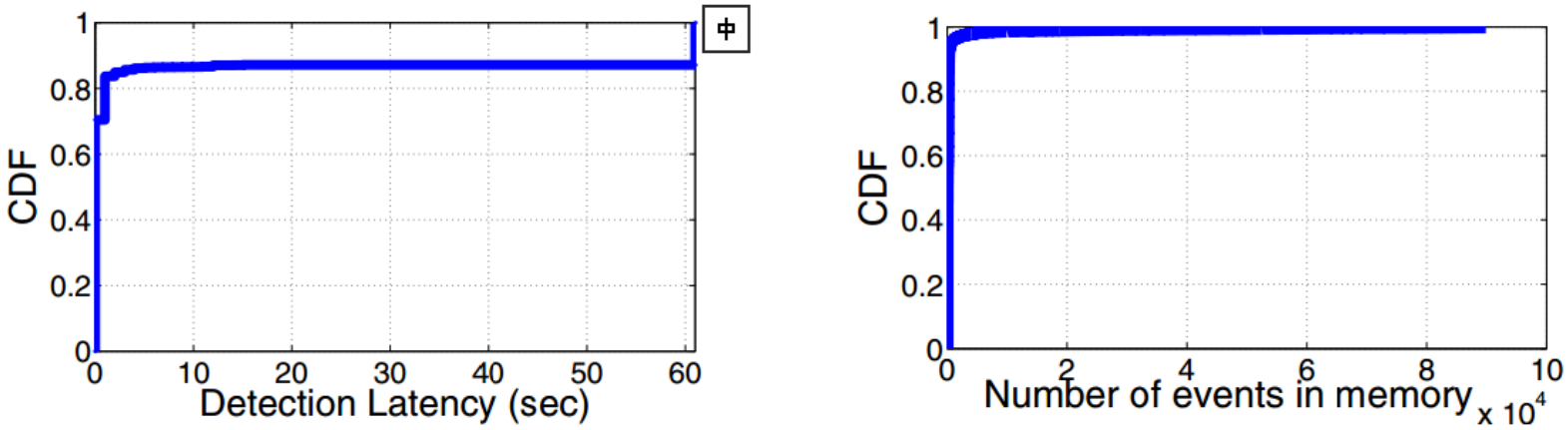


Figure 4. Detection latency and number of events kept in detector's buffer

Online Detection - Evaluation

D. Comparison to offline results

#	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**

Conclusion

- Propose a general approach to problem detection via the analysis of console logs
- Use source code as a reference to understand the structure of console logs to parse logs accurately
- Use parsed logs to construct powerful features capturing both global states and individual operation sequences.
- Use simple algorithms such as PCA yield promising anomaly detection results.
- Adopt a two-stage approach which uses frequent pattern to filter out normal events while using PCA detection to detect the anomalies in an online setting .

Thanks for your attention
Q&A