Stage-Aware Anomaly Detection through Tracking Log Points

Saeed Ghanbari
Department of Electrical and Computer Engineering
University of Toronto
saeed@eecg.toronto.edu

Ali B. Hashemi
Department of Electrical and Computer Engineering
University of Toronto
hashemi@eecg.toronto.edu

Cristiana Amza
Department of Electrical and Computer Engineering
University of Toronto
amza@eecg.toronto.edu

ABSTRACT

We introduce Stage-aware Anomaly Detection (SAAD), a low-overhead real-time solution for detecting runtime anomalies in storage systems. Modern storage server architectures are multi-threaded and structured as a set of modules, which we call stages. SAAD leverages this to collect stage-level log summaries at run-time and to perform statistical analysis across stage instances. Stages that generate rare execution flows and/or register unusually high duration for regular flows at run-time indicate anomalies. SAAD makes two key contributions: i) limits the search space for root causes, by pinpointing specific anomalous code stages, and ii) reduces compute and storage requirements for log analysis, while preserving accuracy, through a novel technique based on log summarization. We evaluate SAAD on three distributed storage systems: HBase, Hadoop Distributed File System (HDFS), and Cassandra. We show that, with practically zero overhead, we uncover various anomalies in real-time.

Categories and Subject Descriptors

D.2.5 [**Diagnostics**]: Testing and Debugging

General Terms

Log, Failure Diagnostics, Anomaly Detection

1. INTRODUCTION

Operational logs capture high resolution information about a running system, such as, the internal execution flow of all individual requests and the contribution of each to the overall performance. Insights gained from operational logs have proved to be critical for finding configuration, program logic, or performance bugs in applications [13, 21, 30–33].

Machine generated logging, at each level of the software stack, and at all independent nodes of a large-scale networked infrastructure, create vast amounts of log data. This makes the main purpose for which logs were traditionally designed, i.e., processing by humans, intractable in practice.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Middleware '14, December 08 - 12 2014, Bordeaux, France
Copyright 2014 ACM 978-1-4503-2785-5/14/12 ...$15.00
http://dx.doi.org/10.1145/2663165.2663319.

To assist users to search for anomalous patterns in the large volume of logs, various automated data mining methods have been proposed [21, 30, 31]. The common feature of these methods is to apply text mining techniques, such as, regular expression matching to infer execution flows from the log messages and expose the anomalous execution flows to the user.

The conventional text-mining methods suffer from two problems: i) they rely on DEBUG-level logging which generates large volumes of data and makes compute and storage requirements excessive, and ii) thread interleaving and thread reuse hide the relationships between log messages and code structure.

The volume of diagnostic log data is large. The footprint of DEBUG-level logging that is essential for diagnosis, is an order of magnitude larger than that of INFO-level logging. For instance, a *Cassandra* cluster of 10 machines, under a moderate workload, generates **500000** log messages per hour (5TB log data per day) with the DEBUG-level logging, a factor of about **2600 times** more than with INFO-level logging. For this reason, text-mining approaches to anomaly diagnosis introduce significant costs for capturing, managing and analyzing the log messages. The common practice is to deploy a dedicated log collection and streaming infrastructure to store the logs, and to transfer them to a different (possibly remote) infrastructure for off-line analysis [2, 8, 17].

In addition to the overhead of conventional log mining methods based on DEBUG-logging, inferring the application's semantics and execution flow from log messages is currently very challenging for two reasons: thread interleaving and thread reuse. Due to thread interleaving, log messages that belong to the same task do not appear in a contiguous order in the log file(s). The thread interleaving problem may be mitigated by printing the thread id with each log message, but this does not solve the second problem, which is thread reuse. A thread might be reused for executing multiple tasks during its lifecycle. Users often need to rely on ad-hoc complex rules to infer the boundaries of tasks from log messages.

In summary, currently users have only two choices: i) to forgo DEBUG-level logging in production systems and give up the crucial information that those logs provide for problem diagnosis, or ii) to pay the high costs of DEBUG-level logging in conjunction with ad-hoc, approximate, and error-prone log mining solutions [21, 30].

In this paper, we introduce Stage-aware Anomaly Detection (SAAD), a low-overhead real-time anomaly detection technique that allows users to benefit from low overhead logging, while still being able to access insights available only with detailed logging. SAAD targets the stage-oriented architecture commonly found in high-performance servers. SAAD tracks the execution flow within each stage by monitoring the calls made to the standard logging library. Stages that generate rare/unusual execution flow, or register unusually high duration for regular flows at run-time indicate

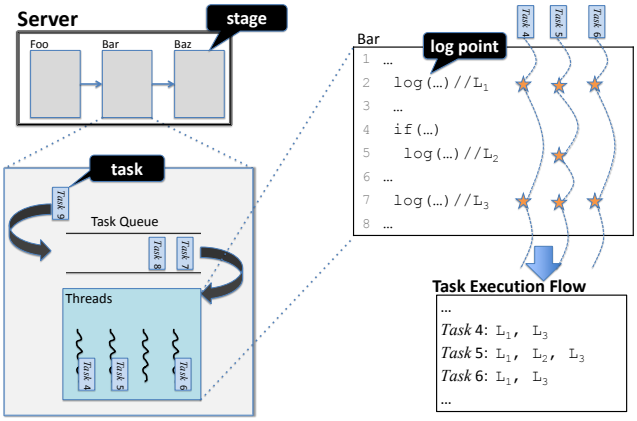


Figure 1: High performance servers execute tasks in small code modules called stages. From the log statements encountered by each task at run-time, we can reason about the task execution flow.

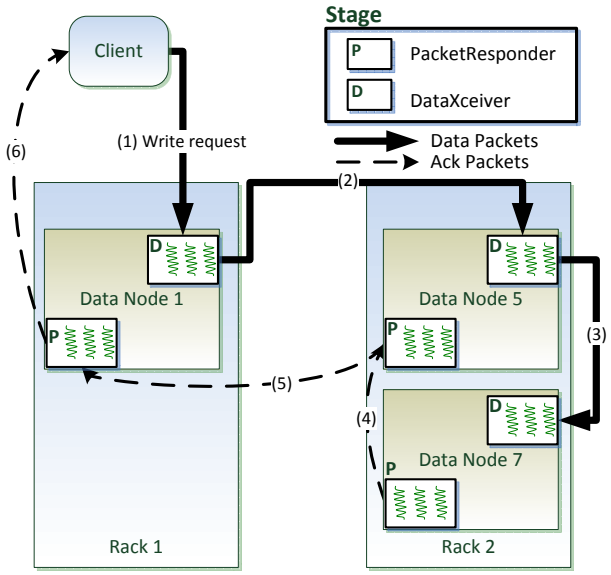


Figure 2: HDFS write operation. In HDFS, a write operation is divided into two tasks on each Data Node; D: receives packets from upstream Data Nodes (or a client) and relays them to the downstream Data Node; P: acknowledges to upstream Data Nodes that packets are persisted by its own node and the downstream Data Nodes.

anomalies.

Figure 1 illustrates the underlying principles of staged architectures. In staged architectures, the server code is structured as small modules, which we call stages, shown as the Foo, Bar and Baz blocks in the Figure. The code of any given stage, e.g., Bar, may be executed simultaneously by several tasks, which are placed in a task queue at run-time, to be executed by separate threads. Each task execution corresponds to a certain flow path taken in the execution of a given stage code by a thread, captured by the logging calls issued by that task.

SAAD leverages all log statements in the code (DEBUG- and INFO-level) as tracepoints to track task execution at run-time. It ignores the content of the log messages, and does not write the log messages to disk. Instead, it intercepts calls to the logger made by a task to record a summary of the task execution. Upon a task completion, the summary of the task execution, which is a tiny data structure of few tens of bytes, is streamed to a statistical analyzer to detect anomalies, in real-time.

Since all tasks pertaining to a stage execute the same code, under normal conditions, each task exhibits statistically repeatable execution flow and duration. The statistical analyzer clusters the tasks based on their similarity to detect rare execution flow and/or unusually high duration.

We minimally instrument the code to insert a few of lines of code to delineate stages in the source code as runtime hints to track the start and termination of tasks. Since the number of stages is limited, the code modification is minimal compared to the magnitude of servers code with tens or hundreds of thousands of lines of code.

The contributions of our SAAD technique described in this paper are as follows:

- **Limiting the search space for root causes:** SAAD leverages the architecture of server codes to limit the space of root causes, by pinpointing to specific anomalous code stages in the code.
- **Detecting anomalies in real-time:** In contrast to existing log analytics that requires expensive offline text-mining, SAAD detects anomalies in real-time, with negligible computing and storage resources.
- **Providing detailed diagnostic data:** SAAD provides detailed diagnostic data by associating the summary of task execution flows with each anomaly.
- **Leveraging existing log statements as tracepoints:** SAAD uses existing log statements in the code as tracepoints to track execution flows. It provides users insights available only with DEBUG-level logging at the same overhead as INFO-level logging.

We evaluate SAAD on three distributed storage systems: HBase, Hadoop Distributed File System (HDFS), and Cassandra. We show that with practically zero overhead, we uncover various anomalies in real-time.

2. MOTIVATING EXAMPLE

We illustrate the stage construct of servers through a real-world example. We then illustrate how the log statements can be leveraged to detect anomalies in tasks.

HDFS Data Node Write. We illustrate the concepts of stage and task in the context of a write operation in HDFS.

Figure 2 shows the execution of a write operation in HDFS, which involves two stages `DataXceiver` (D) and `PacketResponder` (P). Since HDFS uses 3-way data replication, the `DataXceiver` and `PacketResponder` stages might be executed in parallel on 3 Data Nodes. Since many threads may execute the same stage (D and/or P) on the same node, as well as on different nodes, the server architecture just presented offers us opportunities for statistical analysis of many similar task execution flows for detecting outliers per stage, both within each node and across a server cluster.

Our key idea for capturing task execution flow and statistical classification is to track the calls made to the logger from the log points encountered during the execution of a task, as we describe next.

Detecting Anomalies. The intuition behind SAAD is that normal tasks that are instances of the same stage may register different execution flows and/or variability in their duration as part of normal executions e.g., due to being invoked with different input parameters. On the other hand, they are expected to show repeatability in their execution flows.

```
Class DataXCeiver() implements Runnable{
 ...
 public void run(){
 ...
  log("Receiving block blk_"+blockId);  L1
  ...
  while( (pkt = getNextPacket()) ){
    log("Receiving one packet for blk_"+blockId);  L2
    ...
    if(pkt.size() == 0){
      log("Receiving empty packet for blk_"+blockId);  L3
      next
    }
    ...
    log("WriteTo blockfile of size "+ pkt.size());  L4
    ...
  }
  log("Closing down.");  L5
 }
 ...
}
```

Figure 3: Simplified code of HDFS `DataXCeiver` stage.

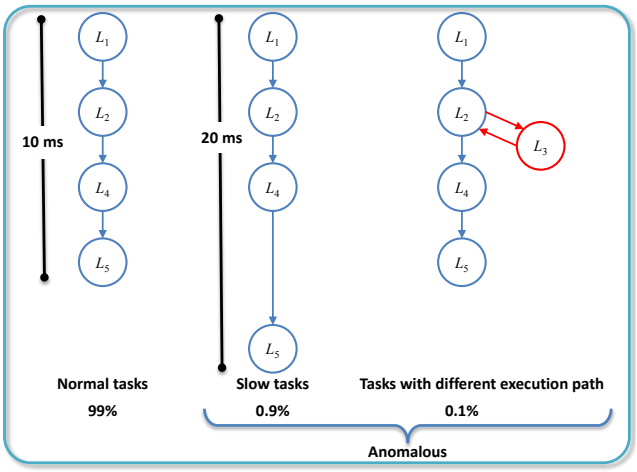


Figure 4: From the log points of tasks executing `DataXCeiver` stage, anomalous tasks with rare execution flow and/or high duration can be detected.

We showcase the key ideas of our real-time statistical analysis on a simplified example of the `DataXceiver` stage on a Data Node in the HDFS write operation in Figure 3.

For the purposes of this example, without loss of generality, we show simplified log patterns of the tasks in this stage in Figure 4. We see that the usual log pattern of the tasks executing this stage, $[L_1, L_2, L_4, L_5]$, occurs 99% of the time with a duration of 10ms. We see that the different log sequence $[L_1, L_2, L_3, L_4, L_5]$ occurs only 0.1% of the time, due to the reporting of an *empty packet* abnormality (associated with L_3). Furthermore, we know that the overall performance of the client write operation depends on the performance of each individual task. The time difference between beginning of a task and the last logging point it encounters is a good indicator for the duration of the task. For example, in Figure 4 we see that the duration of 0.9% of tasks executing `DataXceiver` is 20ms, double the duration of normal tasks, which execute 99% of the time.

This example demonstrates the opportunity to detect tasks with different execution flow and high duration in the executions of a stage.

3. DESIGN

In the following section, we present a high level overview of our Stage-aware Anomaly Detection (SAAD) system. We then present a more detailed description of all components.

3.1 SAAD Design Overview

Stage-aware Anomaly Detection (SAAD) is comprised of two main components: a *task execution tracker* and a *statistical analyzer* as represented, at a high level, in Figure 5.

A task execution tracker running on each node of the server cluster intercepts the execution flow of tasks by registering calls to the logging library per task, and produces a task synopsis at task termination. Task synopses are then tagged with semantic information, such as the stage that the task pertains to, and streamed out to a centralized statistical analyzer.

The centralized statistical analyzer periodically inspects tasks for outliers. An outlier task has rare/new execution flow or registers normal execution flow with higher than normal duration. A stage is considered anomalous if the proportion of outlier tasks to the normal tasks generated by the stage statistically exceeds a limit.

In SAAD, the capturing and streaming of task synopses, and eventually the anomaly detection are done in-memory, without any need to store the task synopses on persistent storage.

In the following, we describe the task execution tracker and the statistical analyzer.

3.2 Task Execution Tracker

The task execution tracker is a thin software layer that sits between the server code and the standard logging library. It tracks the execution flow of each task from the calls the task makes to the logging library, and produces a summary of the task's execution. The task execution tracker i) identifies tasks at runtime, and ii) tracks the execution flow of the tasks.

3.2.1 Identifying tasks

In stage architectures, a task is a runtime instance of a stage that is executed by a thread. We instrument the beginning of each stage code to track association of tasks and threads.

The beginning of a stage in the code is a location where threads start executing new tasks. These locations are identified from the two standard staging models : i) Producer-Consumer model and ii) Dispatcher-Worker model. In the Producer-Consumer model, threads in the producer stage place requests in a queue, and threads in the consumer stage take the request for further processing. The threads in the consumer stage run in an infinite loop of dequeuing a request and executing it. Each request in the queue is handled by a consumer thread, which represents a unique task. Hadoop RPC library and Apache Thrift library [6](used in Cassandra) adopt this model. In this model, the beginning point of a consumer stage is the place where threads dequeue requests.

In the Dispatcher-Worker model, a thread in the dispatcher stage spawns a thread in the worker stage and delegates a task to it. This model is used in cases where a thread defers computation, e.g., by making an asynchronous call, or parallelizing an operation. For instance, the HDFS Data Node `DataXCeiver` in our motivating example uses this model. The beginning point of a worker stage is located at an entry point where threads start executing the code.

3.2.2 Execution tracking

During a static pre-processing pass over all server source code, we assign unique identifiers to all log points, augment each log statement with passing as argument its corresponding log identifier, and record the log points in a log template dictionary.

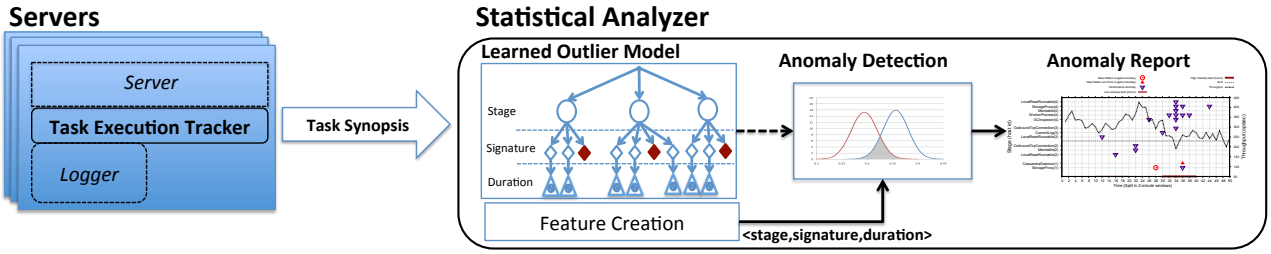


Figure 5: SAAD Overview. SAAD is comprised of *Task Execution Trackers* running on each node and a central Statistical Analyzer. A task execution tracker is a thin layer sitting between the server code and the logger. It tracks execution of tasks by intercepting calls to the standard logger from log statements in the code. At termination of a task, the task execution tracker generates the task execution synopsis. The synopses are streamed to the *Statistical analyzer*, where they are inspected for anomalies in real-time based on a learned statistical model.

Then, at runtime, the task execution tracker *intercepts* the calls to the logger, and registers the succession of log point identifiers encountered during the execution of each task. Each log point is represented by the unique position in a log point vector given by its pre-assigned log point identifier. The tracker further registers the frequency of each log point encountered by a task. Each log point encounter is accumulated in the corresponding entry in the log vector maintained within a per-task, in-memory data structure.

When a task terminates, the task execution tracker generates a synopsis of the task execution. This synopsis is on the order of a few tens of bytes and contains the stage identifier, the task unique id, its start time, and duration, and the frequency of each of the log points.

3.3 Stage-aware Statistical Analyzer

The statistical analyzer detects anomalies from the stream of task synopses at runtime. It detects anomalies that manifest as an increase in outlier tasks in a stage. An outlier is a rare/new execution flow and/or an execution flow with unusually high duration.

Anomalies are reported to the user in a human-understandable way through our visualization tool. The visualization matches outlier tasks with names of stages and semantics associated with the information in log points encountered during execution. In the rest of this section, we illustrate three steps of the statistical analyzer:

1) Feature Creation. From the task synopsis, we first create *feature vectors* for each task sysnopsis. We choose features that capture execution flow and performance aspects of a task.

2) Outlier Detection. Next, we construct a classifier to label tasks as outlier or normal. During runtime, the classifier is used to detect outlier tasks.

3) Anomaly Detection. We apply statistical tests to detect if the proportion of the outlier tasks exceeds a threshold. The outcome of the anomaly detection is presented to users for root-cause analysis.

3.3.1 Feature Creation

We extract two features capturing each task's logical and performance behavior. For capturing logical behavior, we create a signature of the task's execution flow from the distinct log points that it has encountered during execution. For the performance feature of a task, we use the task's duration. We describe each feature in more detail next.

Task Signature. A task signature is a set of unique log points encountered by the task. Each log point in the signature indicates that the task has encountered the log point at least once. For example, the task signature for the normal task in figure 4 is $[L_1, L_2, L_4, L_5]$. The slightest difference in signature is a strong indicator of a difference in the execution flow. More precisely, when signatures of two tasks are different in one or more log points, it means that one of them has executed a part of the code where the other one has not.

Duration. Duration is the time difference between the beginning of a task and the timestamp of the last log point encountered by the task. The duration of a task is a strong indicator of its performance. Faults, such as a slow I/O request, that impact the system's overall performance cause an increase in the execution time of tasks, which is visible in the increased timespans between log points of a task.

The outcome of feature creation for each task is a feature vector: $< id, stage, signature, duration >$ which contains unique id, stage, signature, and duration of the task. The feature vector is used for outlier detection, and anomaly detection, which we explain next.

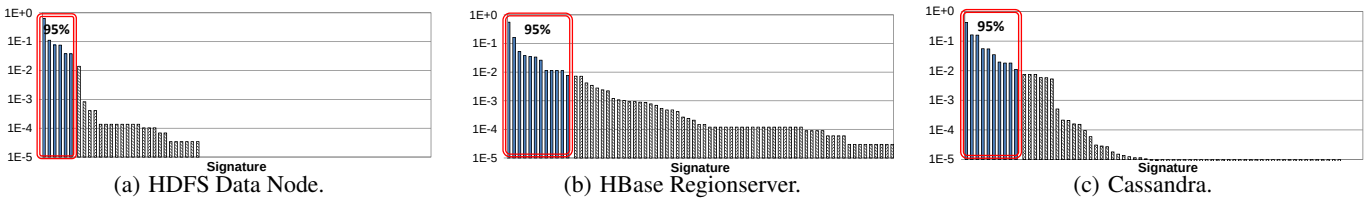
3.3.2 Outlier Detection

For each stage, we classify tasks into normal and outlier based on the task signature and duration. We construct the classifier from a trace of task synopses when the system operates without any known fault.

For each stage, tasks are grouped based on their signatures. We count the number of tasks per signature, and determine the percentile rank of each signature in descending order. Signatures with rank higher than a threshold are considered *flow outliers*. For example, by setting the threshold at the 99th percentile, signatures that account for less than 1% of tasks are considered outliers. The number of signatures is expected to be finite and relatively small because of the finite number of execution flows when the system runs normally. In fact, normal execution flows account for the vast majority of the tasks. We observed this in the systems we studied as shown in Figure 6, where 20% of the signatures account for more than 95% of the tasks in HDFS, HBase, and Cassandra.

Finally, we group tasks with the same stage and signature. For each group, we compute the 99th percentile of the tasks' duration as the performance outlier threshold. The tasks with duration greater than the threshold are considered *performance outliers*.

The duration of tasks with certain execution flows do not register a skewed distribution, which is a prerequisite for being able to determine and select a meaningful outlier threshold. As a result, we cannot accurately classify these tasks into performance outliers. To detect these execution flows, we apply a standard k-fold cross-validation technique. For each signature, we divide the training trace into k equally sized subsets. We construct the outlier classifier from the k-1 subsets and measure the percentage of performance outliers. We repeat the same process for the remaining k-1 subsets. If the average of the percentage of performance outliers for all k sets is significantly higher than the predefined outlier



(a) HDFS Data Node. (b) HBase Regionserver. (c) Cassandra.

Figure 6: Distribution of signatures. Most of the tasks follow a few execution paths. In HDFS Data Node, 6 out of 29, in HBase, 12 out of 72, and in Cassandra 10 out of 68 signatures account for 95% of all tasks.

threshold, we discard the respective signature for the purpose of performance outlier detection.

3.3.3 Anomaly Detection

We define an *anomaly* as a statistically significant increase of outlier tasks per stage. To detect an anomaly, we periodically run statistical tests to verify whether the proportion of outlier tasks exceeds a threshold. We refer to an increase in flow outliers as a *flow anomaly*, and an increase in performance outliers as a *performance anomaly*. In the following, we illustrate the flow and performance anomaly detection in more detail.

Flow anomaly. A stage has a flow anomaly if at least one of two conditions is fulfilled: i) using a *t-test* with significance level of 0.001, the following hypothesis is rejected: the proportion of flow outliers is less than or equal to the proportion of flow outliers observed in the training data, or ii) we observe a new signature that we have not seen during training. A new signature indicates a new execution flow, which can be a strong indication of a fault. For instance, consider a fault that causes a task to terminate prematurely. The premature termination prevents the task to hit some of the log points it would normally encounter, and results in a signature that would have not been seen in the absence of the fault.

Performance Anomaly. For each stage, we group tasks per signature and calculate the proportion of performance outliers. We use a *a t-test* with significance level of 0.001 to verify the following hypothesis: the proportion of performance outliers is less than or equal to the proportion of performance outliers of that signature in the training data. If the hypothesis is rejected, the stage has a performance anomaly.

Anomaly Reporting. We present the detected flow and performance anomalies in a human understandable way for the users and developers to inspect. Each anomalous signature is presented to the user by its stage name, and the list of log templates of its log points. Log templates contain the static portions of the log statements in the code, which reveals the semantic of the execution flow.

4. IMPLEMENTATION

4.1 Task Execution Tracker

We modified `log4j` [1] and added the task execution tracker as a thin layer between the server code and the `log4j` library. `log4j` is the de facto logging library used by most Java-based servers including Cassandra, HBase and HDFS. Our task execution tracker consists of about 50 lines of code.

Tracking task execution records function calls to the logging library from the log statements by each task. We instrument the beginning of each stage with an explicit stage delimiter instruction: `setContext(int stageId)` which hints to the task execution tracker that the thread is about to execute a new task, and passes the stage id. When the `setContext(int stageId)` function is invoked by a thread, the tracker creates a data structure in the thread local storage [5] representing the task. It populates the data structure with stage id, a unique id and the current timestamp. The data structure is also initialized with a map data structure that is used to maintain the id and frequency of log points that will be visited by the task. For every log point encountered by the thread, the map is updated: if the log point is visited for the first time, an entry of log point id will be initialized with value 1 and will be added to the map, otherwise the value associated with log point id will be incremented by 1.

Synopsis. When a task terminates, the tracker generates the execution synopsis of the task from the thread-local data structure. The synopsis is a semi-structured record with the following fields:

```
struct synopsis{
 byte sid; //stage id
 int uid; //unique id per task
 int ts; //task start time (ms)
 int duration; //task duration (us)
 typedef struct {
  short int lpid;//log point id
  int count; //frequency of visit
 }log_points[];
}
```

Determining Task Termination. While the beginning point of each stage is only one and can be identified from the source code, the exit points are multiple and hard to reason about statically from the source code. As an analogy, the beginning point of a function in most programming languages can be easily identified, but the function may have multiple exit points, e.g. `return` statements, or even exceptions which are unknown until runtime. For this reason, the termination of tasks must be inferred at runtime by the tracker. In the case of the producer-consumer model, we infer the termination of a task when the thread is about to start a new task. If a task synopsis data structure is already initialized in thread private storage, it indicates that the thread is finished with the previous task. In the case of the dispatcher-worker model, the termination of a task is inferred from the termination of the worker threads. In Java, we infer termination of a thread through the garbage collection mechanism. When a thread terminates, objects allocated in its private storage become available for garbage collection. Before the garbage collector reclaims space from an object, it calls a special method named `finalize()`. We add proper instructions in this method to generate the task synopsis.

4.1.1 Instrumentation

Stages. We wrote a 40-line Ruby script to parse and analyze the Java source code to identify the beginning of stages to the proper instrumentation. In most cases, the beginning of a stage code corresponds to the place a thread starts executing a code, i.e. `void`

`public run()` method of `Runnable` objects. Instrumenting this place covers: i) all cases of "dispatcher-worker", where a thread is instantiated and a task is delegated to it, and ii) cases of "producer-consumer", where consumer stage is implemented as a standard Java managed thread-pool construct called `Executor` that accept tasks in the from of `Runnable` objects in the input queue.

For other cases of "producer-consumer" that are not based on Executors, we manually add the instrumentation to places in the code where a thread begins a new stage by reading its next request from a request queue. Since, in most cases, Java applications use standard queuing data structures, our script identifies and presents dequeuing points in the source code for manual inspection.

This is a one-time procedure and it is *not* labor intensive because the number of stages is limited. There are 55 stages in HDFS, 38 in HBase Regionservers, and 78 in Cassandra.

Log points. We also instrument log statements to pass an id to identify their location in the code at runtime. We wrote a Ruby script (of about 50 lines of code) which parses the source code and identifies the log statements, and rewrites the log statement with a unique log id as follows. For log info statements we pass the unique log id as a parameter. If there is a check for verbosity in the code, we add the uid as an input to the if statement as follows:

```
if(isDebugEnabled(uid))
 log.debug(...)
```

Our script successfully instrumented 3000+ log statements in HBase, HDFS, and Cassandra in less than one minute. The script also builds a dictionary of log templates i.e., log statements and the information of their respective place in the source code. The log template dictionary is only used for visualization and provided to the user for the purpose of manual root cause diagnosis.

4.2 Statistical Analyzer

We developed the statistical analyzer in R [4]. R is a scripting language with versatile statistical analysis packages. Although R is not designed for high performance computing and it is not multi-threaded, it never became a bottleneck in any of our experiments and allowed for realtime anomaly detection. Our implementation handles streams of task synopses as fast as they are generated, up to the maximum we observed in our experiments which is 1500 task synopses per second. Constructing the statistical model is also efficient: it takes about 60 seconds per host for a trace of 1 hour data of about 5.5 million task synopses. The efficient model building confirms our design decision to limit the computation for training to counting and computing percentiles. During runtime, the computation is extremely light-weight – limited to hash-map operations, to determine if the task's signatures belong to the flow outlier set, simple floating point comparison to determine if the duration falls in the performance outlier region, and t-tests for detecting flow and performance outliers. The synopses are temporarily buffered in memory during model construction. In our experiments, we never exceeded 500MB memory demand to buffer synopses during the model construction.

5. EXPERIMENTS

We evaluate our Stage-aware Anomaly Detection (SAAD) framework on three distributed storage systems: HBase, Hadoop File System (HDFS) and Cassandra. These systems are among the most widely used technologies in the "big data" ecosystem. They consist of distributed components and generate a large volume of operational logs. To avoid generating large volumes of log data, in production environments, the common practice is to set the logging to INFO-level logging. However, by setting the logging to INFO-level logging, we lose valuable information about the execution flows in the system. In this section, we show that with SAAD, we keep the generated logs at INFO-level, while taking advantage of execution flows recorded in task synopses.

We demonstrate that SAAD is effective in pinpointing anomalies in real-time with minimal overhead. The evaluation begins with measuring SAAD's overhead (Section 5.3) with respect to i) task execution tracker runtime, ii) volume of generated monitoring data (task synopsis), and iii) the statistical analyzer computing resource requirements for real-time anomaly detection. Then, in Section 5.4 and Section 5.5, we evaluate SAAD effectiveness in detecting anomalies on Cassandra and HBase/HDFS, respectively. We demonstrate that SAAD narrows down the root-cause diagnosis search by detecting the stages affected by injected faults. Finally, we show the accuracy of SAAD through a comprehensive false positive analysis in Section 5.6.

5.1 Testbed

In this section, we provide the background on HBase, HDFS and Cassandra that we deem necessary to understand and interpret the results.

HBase/HDFS. HBase is a columnar key/value store, modeled after Google's BigTable [9]. HBase runs on top of HDFS as its storage tier. HBase horizontally partitions data into regions, and manages each group of regions by a *Regionserver*. A *master* node monitors the Regionservers and makes load balancing and region allocation decisions. HBase relies on Zookeeper [16] for distributed coordination and metadata management.

HDFS provides a fault-tolerant file system over commodity servers. On each server, a Data Node manages a set of data blocks and uses the local file system as the storage medium. HDFS has a central metadata server called Name Node. Name Node holds placement information of blocks and provides a unified namespace.

Cassandra. Cassandra is a peer-to-peer distributed key/value store, and unlike HBase and HDFS, it does not have a single metadata/-master server. Its distributed data placement and replication mechanism is based on peer-to-peer principles akin to Distributed Hashtables (DHT) [28] and Amazon Dynamo [12]. Cassandra relies on the local file system as storage medium.

Storage Layout of HBase and Cassandra. The storage layout of these two systems is based on Log-Structured Merge Tree [23]. In this layout, writes are applied to an in-memory sorted linked-list called MemTables/MemStores, for efficient updates. Once a MemTable grows to a certain size, it is flushed to disk and stored in a sorted indexed file called SSTable. To guarantee persistence, each update is appended to a write-ahead-log (WAL) and synced to the file system. When the MemTable is flushed, the entries in WAL are trimmed. Flushing a MemTable is called *minor compaction*. As the number of SSTables grows beyond a threshold, they are merged into fewer SSTables in a process called *major compaction*.

5.2 Experimental Setup

We ran our experiments on a cluster of HBase (ver. 0.92.1) running on HDFS (ver. 1.0.3), and Cassandra (ver. 0.8.10). Our cluster consists of nodes with two Hyper-Threaded Intel XEON 3Ghz processors and 3GB of memory, on each node. For the HBase setup, each server hosts an instance of a Data Node and a Regionserver. The HBase Master, HDFS Namenode, and an instance of Zookeeper are all collocated on a separate dedicated host with 8GB RAM. For the Cassandra setup, each Cassandra node runs on a single server.

Workload Generator. To drive the experiments, we use Yahoo! Cloud Serving Benchmark [11](YCSB) (ver. 0.1.4) configured with

Figure 7: SAAD Overhead. Normalized average throughput of HBase and Cassandra with SAAD is compared to their original versions (without SAAD). The error bars (normalized) indicate the variation of the throughput measured every 10 seconds.

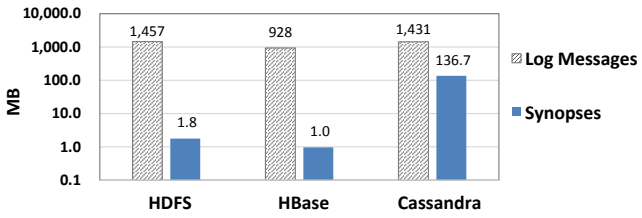


Figure 8: SAAD Reduction in volume of monitoring data. This graph compares the volume of log data generated at DEBUG-level (used in conventional log mining methods) vs. SAAD's task synopses.

100 emulated clients. YCSB is an accepted workload generator for benchmarking NoSql key/value databases, such as HBase and Cassandra. YCSB generates requests similar to real-world workloads.
Workload. In practice, most key/value databases, such as, Cassandra and HBase reside below several layers of caching. Therefore, most requests that reach Cassandra and HBase tiers are write operations. We chose a write-intensive workload mix for our experiments, to make the workload mix resemble the kind of mix that these database systems handle in practice.

5.3 Overhead

5.3.1 Overhead of Task Execution Tracker

In this section, we measure the runtime overhead of the task execution tracker in terms of its effect on the performance of the application and its memory footprint.
Performance Overhead. We compare the throughput of the subject system with SAAD, i.e. instrumented code and the execution tracker, to the original system. For both cases, the logging level is set to INFO-level, which is the default setting in production systems.

Figure 7 compares the throughput of HBase and Cassandra, with and without SAAD. We see that SAAD imposes insignificant overhead on the system.
Memory Overhead. We evaluate the memory footprint of the task execution tracker. The tracker buffers the task synopses (timestamp, stage id, unique id, log frequency vector and the duration), which is 48 bytes on average. Once a task is terminated, its synopsis is sent to the statistical analyzer. In our experiments, the memory usage of the task execution tracker during runtime always remained under a few kilobytes.

5.3.2 Storage Overhead

In SAAD, task synopses are collected in-memory. However, they could be stored for later inspection. Figure 8 compares the volume of logs generated in DEBUG-level logging with the volume of synopses generated for HDFS, HBase, and Cassandra. We see that the volume of task synopses is 15 to 900 times less than the volume of log messages. This highlights the strength of our approach in reducing the storage overhead.

5.3.3 Statistical Analyzer Overhead

We evaluate the statistical analyzer overhead in terms of number of CPU cores needed to process task synopses in real-time. Our current implementation of the statistical analyzer runs on one core.

For comparison, we focus on the text-mining which is the most resource-intensive phase of conventional log analytics methods [30, 32]. These techniques use regular expressions to reverse match the log messages to their corresponding log statement in the code. Regular expression matching over large volumes of data is compute intensive. To speed up the reverse matching process, it is common to use MapReduce framework. We implemented a MapReduce job similar to the one used by Xu et al. [30]. The MapReduce job processed one hour of log data of a Cassandra cluster with 11.9 million log messages (about 1.6GB). It took about 12 minutes of offline-processing on a dedicated cluster of 8 cores.

In contrast, SAAD, by circumventing text parsing through tracking of log points and generating synopses on the fly, requires only one core to produce similar results in real-time.

5.4 Cassandra

In this section, we evaluate SAAD in detecting and pinpointing anomalies in a Cassandra cluster. We show that SAAD can detect the problems that common log monitoring systems which search for error/warning messages are unable to detect.

We conducted several experiments to thoroughly evaluate SAAD on various I/O faults of a Cassandra node. In each experiment, a fault is injected on one of the nodes, to emulate partial failures which are hard to detect due to fault masking.
Failure Model. We injected 4 different faults on the write I/O path based on the following factors:

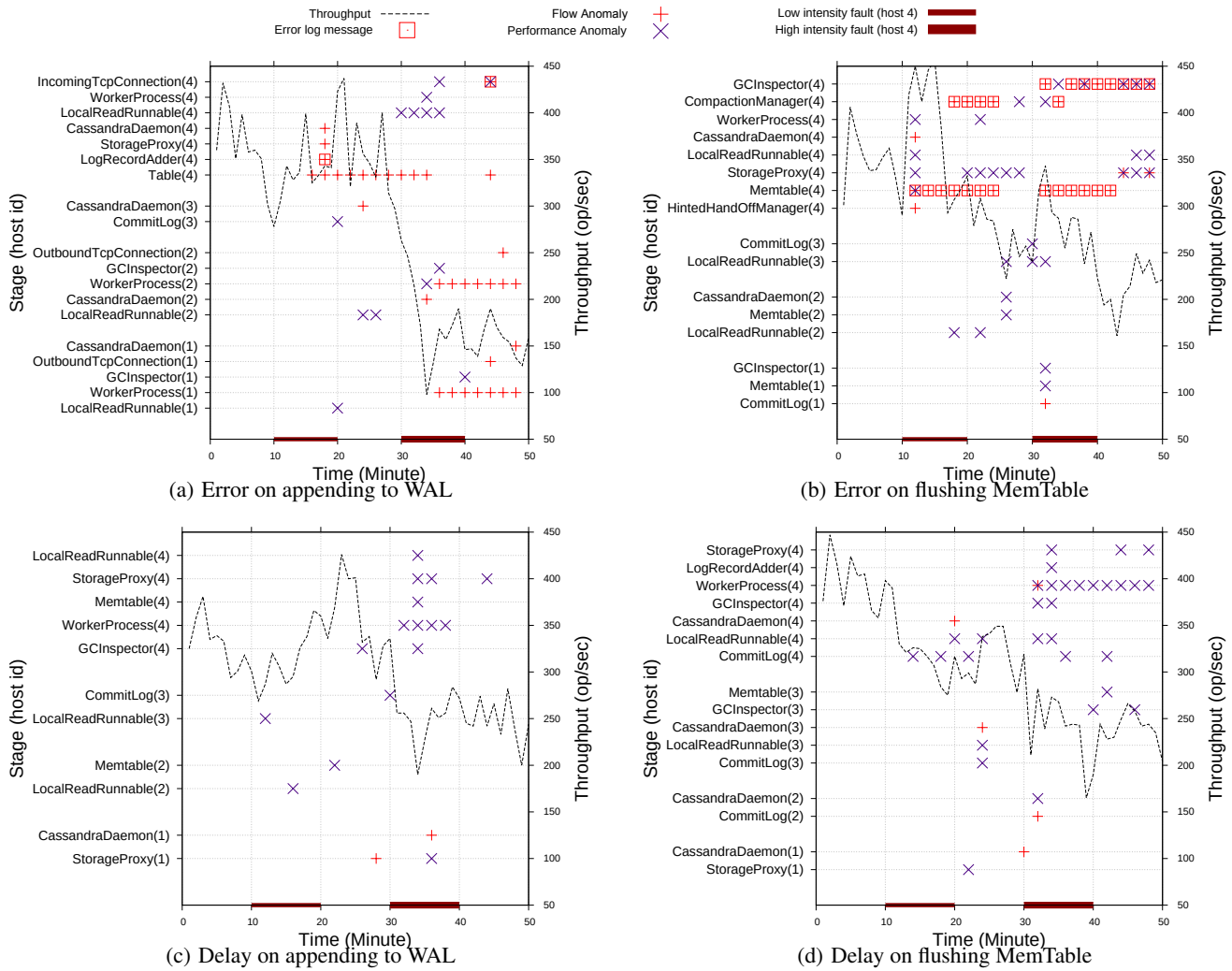
- **I/O activity.** Cassandra has two major types of I/O activities related to MemTables and Write-Ahead-Log (see section 5.1).
- **Type.** We applied two fault types: error and delay faults. In an error fault, we fail I/O requests, and in a delay fault, we pause I/O request for 100ms. The faults are injected using Systemtap [3].

In each experiment, a fault is injected at two intensity levels, low and high intensity. A low intensity fault affects 1% of I/O requests and a high intensity fault affects 100% of the I/O requests. In each experiment, we inject the low-intensity fault at minute 10 for a period of 10 minutes. Then, at minute 30, we inject the high-intensity fault for a period of 10 minutes. We show the performance and flow anomalies per stage in order to highlight the ability of SAAD to detect the relevance of anomalies to the fault. As a baseline, we compare our method with common log monitoring alert systems, where the system alerts the user when an error log is generated.

In these experiments, the statistical model is constructed from a 2-hour trace with 21.7 million task synopses.The model construction (training) took about a minute for each host (4 minutes in total).

5.4.1 Error Faults

Error on appending to WAL. In this experiment, we fail write operations of appending entries to the WAL on host 4. As shown in Figure 9(a), SAAD uncovers several flow anomalies in stage `Table`. The anomalous task signatures in this stage indicate that MemTables are frozen, meaning that the Cassandra node stops applying writes to MemTables.



(a) Error on appending to WAL

(b) Error on flushing MemTable

(c) Delay on appending to WAL

(d) Delay on flushing MemTable

Figure 9: Anomalies per stage in Cassandra. In each experiment, faults are injected on host 4.

Surprisingly, despite the I/O failure, the system generates only one **error log message** at minute 18, and the overall throughput stays unaffected until minute 32, long after the first fault is injected at minute 10. It means that this anomaly cannot be detected by searching for error messages in log data.

To understand why the flow anomaly that SAAD detects in Stage `Table` does not contain an error log message, we show the signatures of the normal execution flow and the anomalous execution flow in Table 1. The normal execution flow includes 4 log statements, while the anomalous execution flow consists of only the first statement reporting that MemTable is frozen. This log statement is not an error. It means that a task must momentarily wait until a lock is released before it can proceed with mutating a MemTable.

However, the injected fault on write I/O to WAL causes a task that is in the middle of applying a mutation and appending to WAL, to get indefinitely stuck and to never release the lock it holds on the MemTable. Consequently, other tasks cannot proceed in mutating the MemTable and terminate prematurely. This premature termination of the tasks is reflected as a rare execution flow, which is uncovered by SAAD.

This highlights the strength of SAAD that uncovers anomalies from execution flows, over conventional log monitoring methods which watch for certain error/warning log messages. The effect of the frozen MemTables on host 4, on which the fault was injected, eventually appears on hosts 1 and 2, which is uncovered by SAAD in stage `WorkerProcess`. Since appending to the WAL and updating MemTables are the mandatory conditions to complete a write, the Cassandra node on host 4 never completes

Table 1: Signature of a normal execution flow and signature of the anomalous execution flow that indicate MemTable is frozen. This anomaly can only be detected as a rare execution flow.

Description of log statements	Normal	Anomalous
MemTable is already frozen; another thread must be flushing it	✓	✓
Start applying update to MemTable	✓	
Applying mutation of row	✓	
Applied mutation. Sending response	✓	

any of the writes that it receives. Once other Cassandra nodes notice that host 4 has become non-responsive, they start delegating writes to random healthy nodes, and request those healthy nodes to retry sending the writes to the failed node (host 4) at a later time. This process of delegation to random nodes for a later retry is called "hinted hand-off". The anomalous flow signatures detected on stage `WorkerProcess` of hosts 1 and 2 indicate that "hinted hand-off" writes for host 4 are timing out.

Eventually, as writes are indefinitely buffered in memory on host 4, the effect of memory pressure becomes visible as a dozen of error messages at minute 44, and shortly after that, the Cassandra process on host 4 crashes.

Error on flushing MemTable. This fault causes error on I/O operations during writing MemTables on host 4. Results are shown in Figure 9(b). SAAD detects flow anomalies in the `MemTable` stage which hint to an I/O problem in writing to MemTables. This stage serializes MemTables and writes them to disk in form of SSTables. We also observe flow anomalies in the `CompactionManager` stage. This stage compacts SSTables. To do so, it reads several SSTables into memory and merge them into one MemTable. Then it writes the MemTable back to disk as a new SSTable.

During the low intensity fault, throughput does not degrade, because only 1% of I/O operations fail. But, during the high-intensity fault, the effect of the fault gradually becomes visible in the performance. During this period, since Cassandra cannot flush the MemTables, memory pressure escalates. This problem is detected in the garbage collection stage `GCInspector` shortly after the high-intensity fault is in effect. We see that even after the fault is lifted, at minute 40, the lingering effect of the memory pressure is detected in the garbage collection stage.

5.4.2 Delay Faults

Delay on appending to WAL. This fault delays write operations to WAL on host 4. SAAD detects several performance anomalies in the `WorkerProcess` and `StorageProxy` stages on host 4 during the high-intensity fault (Figure 9(c)). The `WorkerProcess` stage holds worker threads that handle incoming requests from peer nodes. The execution flow of the outlier tasks in this stage reveals that mutations to rows in MemTables are slowed down. SAAD detects performance anomalies in the `StorageProxy` stage that indicate a slowdown in appending to WAL. Since mutating MemTable and appending an entry to WAL are done transactionally, from these two signatures, the user can reason that a slow down on write to WAL is the root cause of the problem.

Delay on flushing MemTable. This fault slows down I/O operation while flushing MemTables (Figure 9(d)). SAAD detects consecutive performance anomalies in `CommitLog`. Since the `CommitLog` stage trims WAL once a MemTable is successfully flushed to disk, slow-down in this stage hints at a slow-down in MemTable writing to disk.

A MemTable is flushed periodically when it reaches a certain size. The tasks that handle mutations to the MemTable, and happen to add the last entry to a MemTable before it gets full, must flush the MemTables to disk. We see the effect of slowdown as the result of delay in writing MemTables in these tasks as performance anomalies in `WorkerProcess` stage. The execution flow associated with these anomalies indicates that tasks that engage in flushing MemTables are slowed down.

5.5 HBase/HDFS

In this section, we evaluate SAAD in detecting anomalies in a cluster of HBase/HDFS. In this experiment, we injected faults over the course of 3 hours by launching one or more background processes of the following command on *all hosts*: `dd /dev/urandom dummy count=1K count=1M`. This command emulates a disk hog; it consumes the bandwidth of local disks. It also makes several system calls which raise many interrupts and steal CPU cycles from the kernel processes. This causes slowdown of other kernel activities, including network operations. In Table 2 we show the timeline of the injected faults. We began with a low intensity fault and gradually escalated the intensity in the subsequent faults.

Table 2: Description of injected faults on all 4 hosts.

Fault	Span	#of `dd` processes
Low-intensity	8-16	1
Medium-intensity	28-44	2
High-intensity-1	56-64	4
High-intensity-2	116-130	4

Figure 10(a) and Figure 10(b) show the detected anomalies, per stage, in HBase Regionservers and HDFS Data Nodes, respectively. In the figures, we overlay the results with error messages that appear in log messages.

Low-intensity fault: This fault begins at minute 8 and lasts for 8 minutes. We observe some performance anomalies on Regionserver 1 and Regionserver 2. Although the fault is injected on all hosts, we do not observe performance anomalies on other hosts. Because, during this period of time, Regionserver 1 and 2 were receiving more operations than other Regionservers and were already overloaded, the lightweight hog impacted their performance.

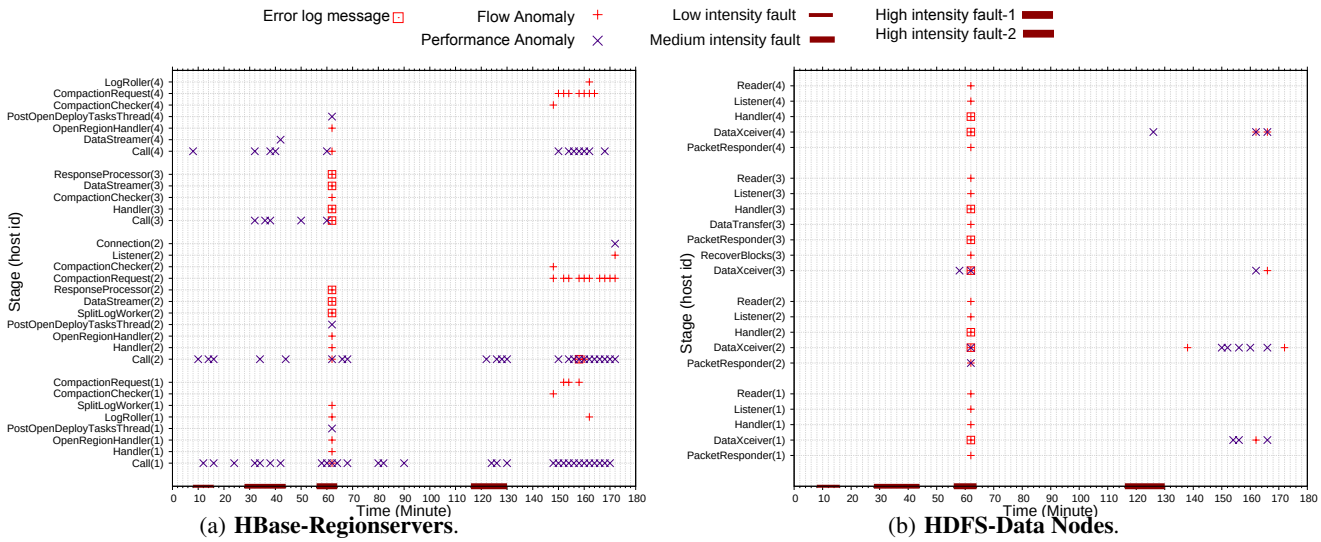
Medium intensity fault: This fault is induced between minutes 28 and 44 and is more intense than the first fault. With this fault, we observed that the performance outliers are significantly increased on all Regionservers. Our model isolates the 'get' RPC calls in stage `Call` as anomalous signatures. This suggests that read operations are slowed down. Since we see no performance anomalies on the Data Nodes, this pattern suggests that the slow-down is most likely caused by CPU contention, rather than I/O slow-down.

High intensity fault-1: This fault is injected between minutes 56 and 64. This fault was the most severe fault we injected into the system which led to a surprising fault scenario; Regionserver 3 unexpectedly crashed. As a result of this crash, we see a surge of flow outliers on all Regionservers and Data Nodes. Our further investigation uncovered that an unfixed bug in the HDFS client library was the cause of the crash, which we explain next.

Bug: premature recovery termination. Our model isolated task signatures that indicated that Regionserver 3 engaged in a repetitive cycle of sending recovery requests to a Data Node to recover a block that it believes to be corrupted. Despite the fact that the Data Node indicates to the Regionserver it is already in the middle of recovery, the Regionserver misinterprets the response as an exception and repeats the recovery request. The anomaly in block recovery is detected as a flow anomaly in the `RecoverBlocks` stage on Data Node 3 (Figure 10(b)).

In cases where the recovery is requested for a block that contains the Regionserver write-ahead-log data, the Regionserver stops processing any write requests (as a rule to guarantee persistence) until the recovery is confirmed on all the Data Nodes that hold a replica of the block. This repetitive cycle eventually leads to exceeding the number of allowed retries and causes the Regionserver to crash. This bug surfaces when Data Nodes become slow in responding to requests due to a high load or a lengthy garbage collection. In our case, the injected hog was the cause of the Data Node slowdown.

High intensity fault-2: This fault is injected between minutes 116



(a) **HBase-Regionservers**. (b) **HDFS-Data Nodes**.

Figure 10: Anomalies per stage in HBase Regionservers and HDFS Data Nodes.

and 130. Like the previous fault, the intensity of this fault was high. We were surprised that the increase in the outlier signals on most servers was not as severe as for the previous fault. We noticed that during this fault, unlike the previous ones, we had very few 'log sync' tasks (in Stage `Handler`) on Regionservers, which are in charge of flushing write-ahead-logs to HDFS. This suggests that few write operations occurred during this period of time, and in fact most performance anomalies were read operations, not write operations. After more investigation, we uncovered a hard-coded misconfiguration in the workload generator, the YCSB emulator version 0.1.4. YCSB configures its HBase client to batch 'put' operations on the client side and to periodically send them in one single RPC call. This artificially boosts performance of write operations, at the expense of delaying writes on the client side. The writes were persisted on Regionservers only after a significant lag of about 9 minutes on average. It must be noted that batching put operations violates the benchmark specifications.

Major Compaction. Close to the end of our experiment, we observed an unexpected spike of outliers on the Regionservers and Data Nodes around minute 150. Our model detects flow outliers in the `CompactionRequest` and `CompactionChecker` stages on the Regionservers. These stages are in charge of performing major compaction operations, where versions of key/values that are stored in separated files (SSTables) are consolidated into fewer files. Since, during this operation, Regionservers issue many I/O requests to HDFS, we also observe performance and flow anomalies in the `DataXCeiver` stage on all Data Nodes. The `DataXCeiver` stage is responsible for handling write operations in HDFS. This is a case of false positive where a legitimate but rare activity is misidentified as an anomaly. In this case, our system could have avoided the falsely detected flow anomaly, if the trace used to construct the statistical model had had at least one case of major compaction.

5.6 False Positive Analysis

In this section, we empirically evaluate SAAD with respect to false positives. In a distributed system with complex dependencies between components, numerous external and internal factors such as network congestion, thread scheduling, and I/O scheduling, may inherently cause transient slowdown or change in execution flows. These anomalies are detected and reported by SAAD. This inherent variability poses challenges in evaluating SAAD, because discerning anomalies caused as a result of a fault (true positives) from anomalies as a result of mis-identification (false positives) is not trivial. Moreover, some of the false positives may be due to unknown causes in the platforms studied; so the best we can do is to provide an upper bound on the potential false positive rates that SAAD signals, which we could find no known cause for.

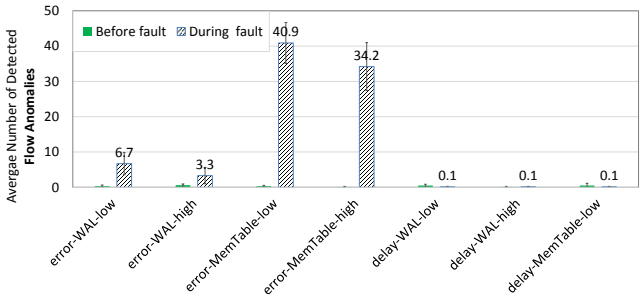
We conduct several controlled experiments for this purpose. Each controlled experiment compares the number of anomalies detected by SAAD with versus without fault injection under otherwise identical experimental conditions. We evaluated SAAD extensively on our Cassandra cluster for 7 different fault types, one per experiment, as shown in Table 3. For statistical significance, we repeat each experiment 10 times. In each run, Cassandra is initialized with a baseline data set. We let the system run 30 minutes to reach stable performance. In the next 30 minutes, we let the system run without injecting any fault. The anomalies detected in this period are, therefore, the result of natural variability in the system, or unknown bugs. We call these anomalies *false positives*. We inject the fault during the third 30 minutes of the experiment. Then, we compare the number of detected anomalies for the periods before and during the presence of the fault.

Flow Anomalies. Figure 11(a) shows the average number of flow anomalies detected before and during the fault injection. The number of flow anomalies detected during the presence of *error faults* (as opposed to *delay faults*) is an order of magnitude higher (by a factor of 10 to 60 times) than before the fault injection. Therefore, filtering out spurious false alarms can be easily added. In addition, the total number of false positives of flow anomalies in all the 70 runs (7 experiments, 10 runs each) is only 54. In other words, the mean time between flow false positives is 38 minutes.

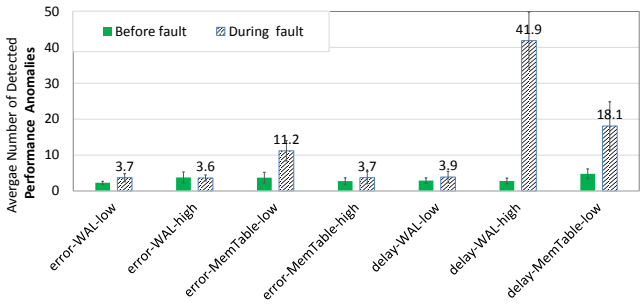
Performance Anomalies. Figure 11(b) shows the average number of performance anomalies detected before and during the fault injection. The number of detected performance anomalies during the presence of WAL-delay-high and MemTable-delay-low faults substantially increases (by a factor of 3 to 8 times). Anomalies do not increase during the presence of the low intensity delay fault,

Table 3: **Empirical validation. We inject 7 faults on the write path of a Cassandra node.**

Name	I/O Activity	Mode	Intensity	Description
error-WAL-low	WAL	Error	Low	Error on 1% of write operations to WAL
error-WAL-high	WAL	Error	High	Error on 100% of write operations to WAL
error-MemTable-low	MemTable	Error	Low	Error on 1% of write operations when flushing MemTable to disk (write to SSTable)
error-MemTable-high	MemTable	Error	High	Error on 100% of write operations when flushing MemTable to disk (write to SSTable)
delay-WAL-low	WAL	Delay	Write	Delay on 1% of write operations to WAL
delay-WAL-high	WAL	Delay	High	Delay on 100% of write operations to WAL
delay-MemTable-low	MemTable	Delay	Write	Delay on 1% of write operations when flushing MemTable to disk (write to SSTable)



(a) **Flow Anomalies**.



(b) **Performance Anomalies**.

Figure 11: The average number of detected anomalies before and during presence of a fault (over 10 runs).

WAL-delay-low, since the fault affects only 1% of writes to write-ahead-logs. We observed 3 false alarms per run, or an average of a 10 minute interval between performance false positives.

5.7 Summary of Results

In this section, we demonstrated that SAAD

- substantially reduces the storage overhead of monitoring data,
- uncovers faults that are not visible in the standard production logging level (INFO-level), with near-zero runtime overhead,
- pinpoints the stages that best explain the source of a fault,
- uncovers hidden patterns in terms of task signatures instead of isolated log messages, which is effective in understanding the meaning of anomalies,
- assists users to avoid the hazard of fault-masking that can lead to a major malfunction,
- proves effective in detecting real-world bugs,
- registers low false positives.

6. RELATED WORK

DISTALYZER [21] provides a tool for developers to compare a set of baseline logs with normal performance to another set with anomalous performance. It categorizes logs into event logs and state logs. Similar to our approach, it uses timing and frequency of event logs to create features that capture the performance of the system. Xu et al. [30] leverage the schema of logging statements in the source code to parse log records. Since the log template of each log record can be determined, a set of rich features, including the frequency of logs with the same type can be extracted from the log sets. They apply principal component analysis (PCA) to detect anomalous patterns. Both methods rely on ad-hoc methods to infer the execution flow from log messages. Unlike our method, these methods do not associate anomalies with the semantic of server code. Fu et al. [15] use Finite State Automata to learn the flow of a normal execution from a baseline log set, and use it to detect anomalies in performance in a new log set. Like our approach, they use timing of log records to build a performance model. Similar to other off-line approaches [19, 20, 27, 31], they use text mining techniques to convert unstructured text messages in log files to log points, which requires expensive text processing. Moreover, the accuracy of their results is highly sensitive to the heuristics used for text processing. Tan et al. [29] also adopt a FSA approach to model the performance of Hadoop MapReduce clusters from the logs. Filtering-based approaches [18, 22] map the sequences of log messages to system administrator alerts in case of incidents.

Sambasivan et al. [25], Pip [24], PinPoint [10] diagnose anomalies by comparing the expected execution flow with the anomalous one. The common denominator of these approaches is detailed fine-grained tracing, which is provided by explicit code instrumentation or thorough distributed tracing enablers such as x-trace [14], Dapper [26] and Magpie [7]. Although, trace-based anomaly detection techniques are more precise than regular log-based approaches, they generate a large amount of monitoring data, which incurs runtime overhead and need to be processed offline. Sherlog [32] infers likely paths of execution from log messages to assist operators in diagnosing root causes for anomalies from anomalous execution paths. Yuan et al. [34] introduce techniques to augment logging information to enhance diagnosability. These approaches are orthogonal to our anomaly detection approach.

7. CONCLUSION

Tracing whole program execution, and detailed logging imply significant run-time overheads, as well as costly post-operational log mining for anomaly detection and diagnosis. At the other extreme, anomaly detection based on production-standard logging verbosity may miss essential clues to anomalies and further complicates the root-cause analysis. In this paper, we propose a sweet-spot design which gives the best of both worlds and is directly applicable to all server codes that are based on staged architectures. We use

minimal instrumentation of server code to mark stages, and track execution flows during runtime from existing log points. We introduce a light-weight real-time statistical analyzer that detects flow anomalies that manifest in the execution flow, as well as performance anomalies that affect the execution time of a system. We validated our anomaly detection technique on three widely-used distributed storage systems, HBase, HDFS and Cassandra. We showed that our system can accurately detect flow and performance anomalies in real time.

References

[1] Apache Log4j. http://logging.apache.org/log4j.

[2] Splunk. http://www.splunk.com/.

[3] Systemtap. http://sourceware.org/systemtap/.

[4] The R Project for Statistical Computing. http://r-project.org/.

[5] Thread local storage. http://wikipedia.org/wiki/thread-local_storage.

[6] A. Agarwal, M. Slee, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. Technical report, Facebook, 2007.

[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[8] D. Borthakur, J. Gray, J. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache hadoop goes realtime at facebook. In *SIGMOD*, 2011.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.

[10] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. Path-based failure and evolution management. In *NSDI*, pages 309–322, 2004.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[12] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[13] C. Ding and K. Kennedy. Improving cache performance of dynamic applications with computation computation and data layout transformations. In *PLDI99*, 1999.

[14] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.

[15] Q. Fu, J. Lou, Y. Wang, and J. Li. Execution anomaly detection in distributed systems through unstructured log analysis. In *ICDM*, pages 149–158, 2009.

[16] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, 2010.

[17] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *NetDB*, 2011.

[18] Y. Liang, Y. Zhang, A. Sivasubramaniam, R. Sahoo, J. Moreira, and M. Gupta. Filtering failure logs for a bluegene/l prototype. In *DSN*, pages 476–485, 2005.

[19] S. Ma and J. L. Hellerstein. Mining partially periodic event patterns with unknown periods. In *ICDE*, 2001.

[20] A. Makanju, A. Zincir-Heywood, and E. Milios. Clustering event logs using iterative partitioning. In *KDD*, 2009.

[21] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI*, pages 26–26, 2012.

[22] A. Oliner and J. Stearley. What supercomputers say: A study of five system logs. In *DSN*, pages 575–584, 2007.

[23] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[24] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI*, 2006.

[25] R. Sambasivan, A. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI*, 2011.

[26] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, 2010.

[27] J. Stearley. Towards informatic analysis of syslogs. In *Cluster Computing*, pages 309–318, 2004.

[28] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01*, pages 149–160, 2001.

[29] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *ICDCS*, pages 795–806, 2010.

[30] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting large-scale system problems by mining console logs. In *SOSP*, pages 117–132, 2009.

[31] K. Yamanishi and Y. Maruyama. Dynamic syslog mining for network failure monitoring. In *KDD*, pages 499–508, 2005.

[32] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from runtime logs. In *ASPLOS*, pages 143–154, 2010.

[33] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *OSDI*, 2012.

[34] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *ASPLOS*, 2011.