Semantic-aware Workflow Construction and Analysis for Distributed Data Analytics Systems

Aidi Pi, Wei Chen, Shaoqi Wang, Xiaobo Zhou
University of Colorado, Colorado Springs
{epi,cwei,swang,xzhou}@uccs.edu

ABSTRACT

Logging is a universal approach to recording important events in system workflows of distributed systems. Current log analysis tools ignore the semantic knowledge that is key to workflow construction and analysis. In addition, they focus on infrastructure-level distributed systems. Because of fundamental differences in log features, they are ineffective in distributed data analytics systems.

This paper proposes IntelLog, a semantic-aware non-intrusive workflow reconstruction tool for distributed data analytics systems. It is capable of building hierarchical relationships between components and events from logs generated by the targeted systems with little or even no domain knowledge. Leveraging natural language processing, IntelLog automatically extracts and formats semantic information in each log message, including system events, identifiers, locality information, and metrics values. It builds a graph to represent the hierarchical relationship of components in the targeted system via nomenclature conventions. We implement IntelLog for Hadoop MapReduce, Spark and Tez. Evaluation results show that IntelLog provides a fine-grained view of the system workflows with semantics. It outperforms existing tools in automatically detecting anomalies caused by real-world problems, misconfigurations and system bugs. Users can query the formatted semantic knowledge to understand and further troubleshoot the systems.

KEYWORDS

log profiling, distributed systems, natural language processing, workflow construction, troubleshooting

ACM Reference Format:
Aidi Pi, Wei Chen, Shaoqi Wang, Xiaobo Zhou. 2019. Semantic-aware Workflow Construction and Analysis for Distributed Data Analytics Systems. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC ’19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3325404

1 INTRODUCTION

Distributed clusters have been commonly used to support growing computational demands. In order to take advantage of distributed clusters, numerous data analytics systems are designed for clusters of distributed machines. For example, MapReduce [16] and Spark [37] are two popular general-purpose data analytics systems that are designed for distributed deployment. TensorFlow [8] is a distributed system for large-scale machine learning jobs.

Understanding and troubleshooting distributed data analytics systems is critical to ensure reliable production environments. Those systems use logging to record important events in execution, including configuration information, events in workflows, system status and error messages. Administrators use logs to understand the underlying systems and perform anomaly analysis. However, manual log analysis is intimidating due to the system complexity. One job request with different configurations can generate log sessions with variable lengths. Numerous components are usually involved when a system responds to a user request. When an anomaly occurs, an administrator needs to understand the complex workflow of various components in order to analyze the root causes, which makes the troubleshooting work difficult and time consuming.

Log analysis tools can generally be classified into two categories: information inference and automatic anomaly detection. Information inference tools [28, 30, 38, 39] help users to understand the targeted system more efficiently but leave the burden of anomaly analysis to users. They require users to have domain knowledge of the systems. To achieve automatic anomaly detection, there are tools that leverage machine learning and automaton methods [18, 26, 36]. Those tools can report potential problems and reduce user efforts in log analysis. However, they focus on infrastructure-level distributed systems (e.g., OpenStack [4] and HDFS [10]) which have log sessions with relatively fixed lengths. More importantly, the existing tools of the both categories do not exploit the semantic knowledge of logs, which however is key to workflow construction and analysis for distributed data analytics systems.

Our work is inspired by the original goal of system logs: *logs are for users to read.* In other words, logs are usually written in natural languages by system designers. Leveraging semantic information in logs for workflow construction and troubleshooting not only provides users a clear view of the workflows of systems, but also reduces the user efforts for manual analysis. In order to extract semantic knowledge from log messages, our intuition is that natural language processing (NLP) should be used to process logs that are written in natural languages.

We propose IntelLog, an NLP-assisted log analysis tool that reconstructs workflows of distributed data analytics systems and automatically pinpoints the erroneous components when anomalies occur. By leveraging NLP approaches, the goal of IntelLog is to provide a workflow graph of a targeted system. IntelLog also relies on the workflow graph to conduct automatic anomaly detection.

However, there are technical challenges to develop NLP-assisted workflow reconstruction and anomaly detection. First of all, it is difficult for existing NLP tools to extract useful information from

HPDC ’19, June 22–29, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3325404

log messages since they are different from free-form text. Secondly, an effective tool should build relationships between objects and events in the targeted systems even if they are not associated with identifiers. Ignoring such log messages can lead to incomplete reconstructed workflow. Finally, one request in distributed data analytics systems can generate logs with various lengths and orders. Currently, tools [18, 36] perform automatic detection in infrastructure-level distributed systems only, of which the same request usually outputs log sequences with short lengths and a relatively fixed order. Thus, they are not effective in data analytics systems.

The novelty of IntelLog lies in that it extracts semantics from system logs via NLP-assisted approaches and it achieves workflow reconstruction and anomaly detection for distributed data analytics systems. Its contribution lie in the following three aspects. Firstly, IntelLog leverages the Part-of-Speech (POS) analysis approach to extract entities (both objects and events) from log messages. Such an approach is also helpful in distinguishing identifiers from numeric values since both of them can be represented by numeral strings. Using the results from POS analysis, IntelLog is able to extract operations done by entities via parsing the sentence structure of a log message. After the parsing, the log keys are transformed to a key-value like storage called Intel Keys. We call a log message corresponding to an Intel Key as an Intel Message.

Secondly, IntelLog builds a Hierarchical Workflow graph (HW-graph) that represents the lifespans of entities and execution of a targeted system. In order to build the HW-graph, IntelLog first analyzes the nomenclature of entity names and groups correlated entities together. Then, it analyzes the lifespans of the entities in each group and builds the hierarchical relationship. For data analytics systems, workflows in an entity group usually output identifiers to distinguish from each other. Thus, we further refine our design by leveraging the identifiers to check the subroutines. The HW-graph that abstracts the workflow helps users understand task execution and analyze causes of anomalies.

Last but not least, IntelLog uses the HW-graph to perform automatic anomaly detection. It obtains the relations of each entity group as well as the log sequences in each group from normal execution. In the anomaly detection phase, IntelLog checks the log messages against the HW-graph built from normal execution and tries to reconstruct a full graph. If an incomplete graph is built, IntelLog can pinpoint the abnormal components. For an unexpected log message, IntelLog also extracts the information from it and transforms it to an Intel Message. Note that IntelLog does not directly find the root causes that can be related to multiple factors in a distributed system. Instead, pinpointing the abnormal components assists users to narrow down the reasons that cause the anomaly.

We evaluate IntelLog in three general-purpose data analytics systems, MapReduce [16], Spark [37] and Tez [31]. Experimental results show its NLP-assisted approaches achieve high accuracy in entity extraction. IntelLog reconstructs hierarchical relationships between components in the targeted systems and the operations done by the components in a fine-grained manner. For workflow reconstruction, we compare IntelLog with a existing tool Stitch [39], and show that IntelLog is able to present more abundant workflow information to users. Evaluation shows that IntelLog reaches a 87.23% precision and 91.11% recall in problem detection of real-world scenarios. Users can query the formatted semantic knowledge to locate the root causes. IntelLog also has the power to detect unexpected anomalies due to inappropriate configurations or system bugs. We compare it with DeepLog [18] and LogCluster [21] and results show that IntelLog achieves better performance of anomaly detection when applied to distributed data analytics systems.

```
1 fetcher # 1 about to shuffle       1 fetcher # * about to shuffle
  output of map attempt_01              output of map *
2 [fetcher # 1] read 2264 bytes      2 [fetcher # *] read * bytes
  from map-output for attempt_01        from map-output for *
3 host1:13562 freed by               3 * freed by
  fetcher # 1 in 4ms                    fetcher # * in *
red: entity  blue: identifier  green: value  purple: locality
```

Figure 1: A real-world log snippet of MapReduce.

In the following, Section 2 introduces logging and IntelLog overview. Section 3 focuses on NLP-assisted information extraction. Section 4 describes the HW-graph. Sections 5, 6, and 7 describe implementation, evaluation and discussions of IntelLog, respectively. We present related work in Section 8 and conclusion in Section 9.

2 PRELIMINARIES AND OVERVIEW

2.1 Preliminaries

A log message has a constant text field and variable fields that record information such as identifiers, amount of processed data, time spent, localities. Correspondingly, the text field and the variable fields are outputs by the constant string and variables in the printing statement in the source code. A log printing statement can be abstracted as a *log key*, in which the constant field is unchanged while the variable fields are represented by asterisk (*). Log keys is widely used by previous studies for the purpose of log analysis [18, 36, 39]. We use Spell [17] to extract log keys from log messages. It consumes raw logs generated from the systems and adopts a longest string matching algorithm to identify the log keys.

However, a log key is a coarse grained abstraction of log messages since it does not distinguish the variable fields. IntelLog parses the log keys and analyzes the meanings of the variable fields by NLP-assisted approaches (§3). We use a simplified real-world log snippet from MapReduce, shown in Figure 1, to introduce the terminologies that will be used in this paper. The log snippet describes the *subroutine instance* of a `fetcher` that gets data from a remote node. The instance is a log sequence consisting of three log messages on the left side. Their corresponding *log keys* are on the right side. For each log key, IntelLog distinguishes four categories of information from the variables fields including *entities*, *identifiers*, *values* and *localities* marked in colors in Figure 1. Then, the log keys are transformed to Intel Keys with key-value pairs representing each category of information. The sequence of the Intel Keys are called a *subroutine* in the HW-graph.

2.2 Log Features

We observe three log features that offer opportunities to take advantage of NLP-assisted approaches for workflow reconstruction and anomaly detection via log analysis.

First, we find that using natural languages to record system events is common in distributed systems. In this paper, we define that a log message is written in a natural language if it contains

Table 1: Lines and percentages of natural language logs.

System	NL logs	total logs	% of NL logs
Spark	1,286,159	1,286,159	100%
MapReduce	599,103	549,922	91.8%
Tez	112,478	103,609	92.2%
Yarn	156,196	159,995	97.6%
nova-compute[1]	73,318	73,318	100%

at least one clause. We analyze over 300MB logs generated from three popular data analytics systems (MapReduce, Spark and Tez), a cluster resource management framework (Apache YARN) and a component of a cloud infrastructure (nova-compute in OpenStack [4]). Table 1 shows the number of lines and percentages of natural language logs in the five systems, which indicate that most of the logs are written in natural languages. Other logs usually record status of the systems such as access control and resource usages. The log feature inspires us that NLP-assisted approaches can be applied to analyze log messages generated by systems. However, log messages have more identifiers and domain-specific words compared to free-form text written by human. This fundamental difference makes it challenging to apply NLP analysis on logs.

Second, although logs contain domain-specific words, we are able to find the relationship between correlated entities by analyzing the nomenclature. To be specific, correlated entities usually share common sub-phrase in their names. Thus, users know those entities are closely related when reading the logs. For example, Spark uses `block` to store data and uses `BlockManager` to do block management. Finding such entities gives users a clear view of the correlated components in the system. Nevertheless, it is time consuming for users to manually find such entities by key word searching due to the large number of logs. Furthermore, it is possible that entity names with a common sub-phrase are not correlated. Therefore, a more subtle algorithm should be used in order to distinguish such entities as well as group correlated entities.

Third, since logs are output by execution of the structured source code, the log sequences may follow specific orders [18]. However, the order is not strict in distributed systems due to concurrency. Currently, existing log analysis tools [18, 21, 36] focus on the order between individual log messages of infrastructure-level distributed systems such as OpenStack [4] and HDFS [10]. However, they do not exploit the semantic knowledge in logs. The log sequences generated by one request in those systems are usually of a short and fixed length since the number of operations in the execution is deterministic. For example, OpenStack has eight most frequently used requests, each generating a fixed-length log sequence with an average length of nine [36]. This feature facilitates workflow reconstruction for infrastructure-level distributed systems.

On the contrary, logs generated by prevalent distributed data analytics systems (e.g., MapReduce, Spark and Tez) are usually with various lengths and interchangeable orders. Different data sizes and configurations cause various log sequence lengths while parallel executions cause interchangeable orders. Our experiments show that the log sequence lengths of these systems range from tens to thousands (§6.4), which makes existing log analysis approaches ineffective in distributed data analytics systems. By leveraging the nomenclature of entity names, IntelLog is able to find the ordering and hierarchical relationships between groups of entities. As a result, users can have both an overview and a detailed look of workflows in the systems.

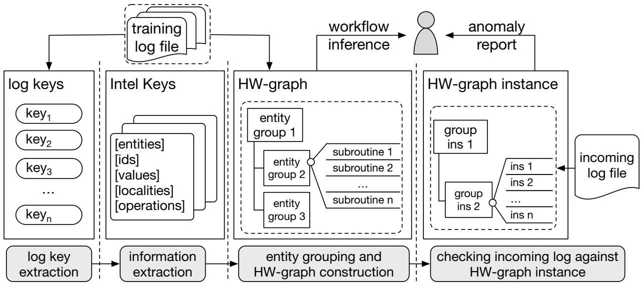


Figure 2: The Overview of IntelLog.

2.3 IntelLog Overview

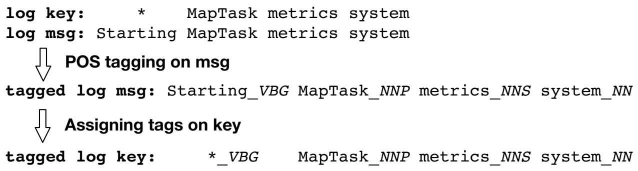
Figure 2 shows the overview of IntelLog. Its first stage, similar to other tools, extracts log keys from log files generated by the targeted systems. The second and third stages build Intel Keys and construct workflows of the systems represented by HW-graphs. The fourth stage consumes newly incoming logs and automatically report anomalies to the users.

Entity extraction. The second stage extracts and distinguishes fields in log keys using two NLP approaches: POS analysis and sentence structure analysis. The extraction result of this stage is execution information of the systems that includes entities, identifiers, values, localities and operations. The extracted fields from a log key is stored in an Intel Key that consists of a set of key-value pairs so that users can use queries to request data. An Intel Key is an enhanced representation of a log key. The extracted entities are also used in the next stage to reconstruct workflows (§3).

HW-graph modeling. Before building the HW-graph, we group correlated entities based on their names using nomenclature convention. Besides finding the ordering relationships between individual Intel Keys, we construct hierarchical relationships between entity groups. This step is done by finding the lifespan of each entity group. In an entity group, there are multiple subroutines that represent the ordering relationships between Intel Keys. A HW-graph represents the workflow of a targeted system, which assists users to understand the system. (§4.1)

Anomaly detection. IntelLog instantiates a HW-graph instance when a system starts a new session. A session is generally an execution path invoked by a user request. In our case, a session is the execution within one Yarn container (§5). While consuming incoming logs, IntelLog aims to build the graph instance to meet the structure of the corresponding HW-graph. Per its design, IntelLog reports two categories of anomalies to users: 1) unexpected log messages, and 2) erroneous HW-graph instances. Similar to previous studies [18, 36], this step does not directly find root causes

[1] Nova-compute constantly outputs log messages with a fixed format that reports the current node resource usage regardless whether there are VM requests. We eliminate such messages and only calculate log messages that are related to VM requests.



```
log key:     *    MapTask metrics system
log msg: Starting MapTask metrics system
tagged log msg: Starting_VBG MapTask_NNP metrics_NNS system_NN
tagged log key:     *_VBG    MapTask_NNP metrics_NNS system_NN
```

Figure 3: POS tagging on a log key.

of anomalies. It reports the affected components and entity group, which significantly reduces the user effort for root cause analysis.

3 INFORMATION EXTRACTION

After collecting system logs, we use Spell [17] to obtain the log keys of the logs. This section describes how we extract information from the log keys and transform them to Intel Keys. The extracted information includes entities, identifiers, values, localities and operations recorded in the log keys. The former four categories of information is extracted by a POS tagging based approach while the last one is extracted by a sentence structure parsing based approach. The information extraction phase transforms each log key to an Intel Key. The log message matching the corresponding Intel Key can be easily stored in databases for queries.

The first step in the information extraction phase is POS analysis. Existing POS tagging tools are widely used in academia and industry, such as OpenNLP [3], Stanford POS tagger [33] and NLTK [11]. However, POS analysis for log keys is slightly different from that for free-form text. Since a log key is an abstraction of log messages, it contains variable fields that is represented by asterisk (*). If we directly feed log keys to a POS tagger, the results will not be accurate. Thus, for each log key, we take a corresponding sample log message and feed it as the input of the POS taggers. The output POS tags are used for the log key. Figure 3 shows an example of this process. The log key ‘`* MapTask metrics system`’ is an abstraction of the log message ‘`Starting MapTask metrics system`’. Apparently, it is inappropriate to feed the log key as the input to the POS tagger. In this case, we use the sample log message as the input. The result of POS tagging is a log message in which each word is tagged with its part-of-speech. Finally, we assign the corresponding words in the log keys with the POS tags in the sample log message. We use the Penn Treebank tag set [24] as our POS marks.

3.1 POS Tagging Based Extraction

Entity extraction. The step is based on a theory in a previous study [20], which postulates that over 97% of terminological entities only consist of nouns and adjectives and thus can be matched by a list of seven POS patterns. Furthermore, in system logs, entities can also be a single-word noun. We use Penn Treebank to present these patterns as shown in Table 2. Note that patterns ‘JJ JJ NN’ and ‘JJ NN NN’ have no example since we do not observe such patterns in our implementation. We keep these two patterns to make IntelLog extensible to other systems.

Beside the POS pattern matching, we also use a camel-case word filter to find entity names. The intuition is that some entities in logs are also classes defined in the source code whose names follow the camel-case naming format convention. Such entities are usually correlated to other entities such as ‘`MapTask`’ and ‘`map output`’. This filter separates a camel-case word into a phrase. In this example, ‘`MapTask`’ is transformed to ‘`map task`’. After we extract the entity phrases, we lemmatize them to their singular forms. Note that users can define their own filters for other naming format conventions.

Table 2: Patterns to match phrases and the corresponding Penn Treebank presentation. Due to the limited space, ‘NN’ includes four noun tags: ‘NN’, ‘NNS’, ‘NNP’ and ‘NNPS’.

Patterns	Penn Treebank Pst.	Examples
noun	NN	task
adjective noun	JJ NN	remote process
noun noun	NN NN	event fetcher
adjective adjective noun	JJ JJ NN	-
adjective noun noun	JJ NN NN	-
noun adjective noun	NN JJ NN	cleanup temporary folders
noun noun noun	NN NN NN	map completion events
noun preposition noun	NN IN NN	output of map

Table 3: UD relations for operation extration.

Element	Relations	Descriptions
predicate	ROOT	a relation indicating the root of the sentence
	xcomp	an open clausal complement of a verb or an ajective
subj-entity	nsubj	a nominal subject of a clause
	nsubjpass	a passive nominal subject
obj-entity	dobj	a direct object of a verb
	iobj	an indirect object of a verb
	nmod	a nominal modifier of a clausal predicates

Locality extraction. We define a set of patterns to capture commonly used locality information in distributed systems. These patterns include: 1) host names, 2) IP addresses and ports, 3) local directory paths, and 4) distributed file system paths. Besides, users can define new patterns when applying IntelLog on their own targeted systems.

Identifier and Value extraction. Both identifiers and values are represented by variable fields in a log key. Previous studies either manually define identifier patterns [36] or do not distinguish these two kinds of variables [18]. We apply four heuristics one after another on a variable field, which accurately recognize identifiers and values. First, we filter out variable fields that have verb POS tags and locality information recognized in the previous steps. Second, we categorize a field as a value if it is followed by a unit, such as ‘`12 MB`’ and ‘`5 ms`’. Third, we categorize a field as an identifier if the field is mixed with letters and numbers, such as ‘`attempt_01`’. Finally, for the fields consist of only numbers, we check the POS tag of the word prior to the field. We categorize the field as an identifier if the POS tag represents a noun. Otherwise, the field is a value.

3.2 Structure Parsing Based Extraction

Operation. The idea is that an operation is usually indicated by a predicate in a log key, and the entities related to the predicate are either the source or the target of the operation. We use universal

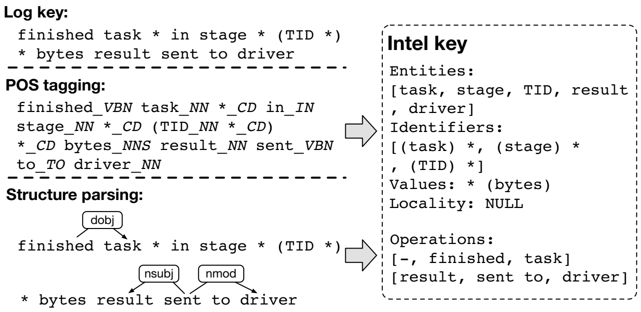

Log key:
finished task * in stage * (TID *)
* bytes result sent to driver
POS tagging:
finished_VBN task_NN *_CD in_IN
stage_NN *_CD (TID_NN *_CD)
*_CD bytes_NNS result_NN sent_VBN
to_TO driver_NN
Structure parsing:
dobj
finished task * in stage * (TID *)
nsubj
nmod
* bytes result sent to driver
Intel key
Entities:
[task, stage, TID, result
, driver]
Identifiers:
[(task) *, (stage) *
, (TID) *]
Values: * (bytes)
Locality: NULL
Operations:
[-, finished, task]
[result, sent to, driver]

Figure 4: Process of transforming a log key to an Intel Key.

dependency (UD) relations [27] to denote the structure of a log key. For simplicity, we define an operation as a 3-tuple: {`subj-entity, predicate, obj-entity`}. Since most entities are nouns, the sub-entity and obj-entity can be extracted by checking whether a word has a specific relation with the predicate. We carefully choose 7 relations out of a total of 40 UD relations. Table 3 describes these relations. These relations indicate a predicate and the entity related to the predicate.

3.3 A Summary Example

We take a log key in the critical path of Spark jobs as an example to demonstrate the process of the information extraction approach as shown in Figure 4. This log key records the task finish event and the amount of data sent to the driver. After the POS tagging phase, each word in the log key is tagged with a POS mark. IntelLog extracts five entities (but omit 'bytes' since it is a unit), three identifiers and one value. The structure parsing technique allows us to obtain the relations between words. Two operations are extracted according to the relations in Table 3.

The resulted Intel Key is shown on the right side of Figure 4. When a newly incoming log message matches the Intel Key, the variable fields represented by asterisks are replaced by the actual values in the log message. As a result, the log message is transformed to an Intel Message. Compared to a log message, an Intel Message is a more structured representation of logs since it structurizes both the text formats and the contents of logs. Further, an Intel Message can be considered as a collection of key-value pairs. It naturally fits in the storage structure of time series databases [1, 5] and can be utilized by other profiling tools [28].

4 MODELING AND ANOMALY DETECTION

IntelLog builds the HW-graph to present the workflow of a targeted distributed system. First, entities are grouped by their names. We identify the hierarchical relationships between groups by checking their lifespans. In each group, we leverage the log orders and identifiers to build subroutines.

In the anomaly detection phase, IntelLog tries to reconstruct a HW-graph instance from the incoming logs and check it against the HW-graph built for the system. IntelLog reports two kinds of potential anomalies to users: 1) unexpected log messages, and 2) incomplete subroutines.

4.1 Workflow Reconstruction

Entity grouping. In a targeted system, entities are usually correlated with each other. However, such entities are not always associated with the same identifiers. Thus, Stitch [39], which constructs workflows solely based on identifiers, only presents workflows in a course-grained manner. Moreover, the identifier names only contain limited semantic knowledge, leaving a large amount of information unexploited.

Algorithm 1 Grouping entities.

```
Input: a list of all extracted entities in ascending order by the # of words: $\mathcal{E}$;
a dictionary $\mathcal{D} < group, \mathcal{E}_g >$ that maps a common phrase $group$ to a set of entities $\mathcal{E}_g$;
a dictionary $\mathcal{D}_r < entity, \mathcal{G} >$ that is a reverse key index of $\mathcal{D}$, mapping an entity to a set of groups $\mathcal{G}$;
$\mathcal{D} \leftarrow \{\}$;
for all $e$ in $\mathcal{E}$ do
    $grouped \leftarrow false$;
    for all $(group, \mathcal{E}_g)$ in $\mathcal{D}$ do
        $com\_phrase_l \leftarrow$ LongestCommonPhrase($group, e$);
        if $com\_phrase_l \neq NULL$ then
            $\mathcal{E}_g \leftarrow \mathcal{E}_g \cup \{e\}$;
            $group \leftarrow com\_phrase_l$;
            if ! $grouped$ then
                $grouped \leftarrow true$;
            end if
        end if
    end for
    if $grouped = flase$ then
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{e, \{e\}\}$;
    end if
end for
$\mathcal{D}_r \leftarrow$ ReverseIndex($\mathcal{D}$)

function LongestCommonPhrase($G, E$)
    if $G$ has one word or $E$ has one word then
        return the longest common string of $G$ and $E$;
    else if $G$ and $E$ have common last few words then
        return empty string;
    end if
    return the longest common string of $G$ and $E$;
end function
```

In IntelLog, we leverage the nomenclature of entities to group together log keys that contain correlated entities. The intuition is that correlated entities usually share the common sub-phrase in their names. For example, `block`, `block manager` and `block manager endpoint` share a common sub-phrase 'block'. However, some entities are not correlated even they share a common sub-phrase. We find that such phrases usually share last few words that have a general meaning. For example, phrases '`block manager`' and '`security manager`' in Spark share the common sub-phrase 'manager' but they are not tightly correlated.

Algorithm 1 describes the algorithm for grouping entities. It takes a list of extracted entities as an input and maintains a dictionary that maps group names to a set of entities. It compares each entity *e* in the list to each *group* in the dictionary and returns the longest common phrase. If either *e* or *group* only contains one word, the function returns the longest common phrase of these two words. In this case, it is either a one-word phrase or an empty string. Since the one-word phrase is part of the other multi-word phrase, their meanings are correlated. For two multi-word phrases, the function also checks whether the two phrases only have the last few words in common. If so, the function ignores the common phrase and returns an empty string. Otherwise, the function returns the longest common phrase that could contain one or multiple words (Line 23 ~ 30). The idea behind of this step is that the last few words of a system entity usually have general meanings such as 'manager', 'file' or 'output'. The algorithm does not group together entities sharing the last few words since their meanings are not tightly correlated. In theory, it is possible that such entities are correlated. Thus, the algorithm categorizes these entities into different groups, missing the correlation. However, IntelLog can still capture the lifespans of the entity groups during HW-graph construction phase. In practice, we do not encounter such entity groups.

Algorithm 2 Assigning Intel Keys into subroutines.

```
1:  Input: a set of log message sequence in the entity group collected from
    different sessions: $\mathcal{S} < Seq_{log} >$;
2:  a dictionary mapping sets of identifier values to log message sequences:
    $\mathcal{D}_{vl} < \mathcal{S}_v, Seq_{log} >$;
3:  a dictionary mapping sets of identifier types to Intel Key sequences:
    $\mathcal{D}_{ti} < \mathcal{S}_t, Seq_{key} >$;
4:  for all $Seq_{log}$ in $\mathcal{S}$ do
5:      $\mathcal{D}_{vl} \leftarrow \{\{NONE\}, []\}$;
6:      for all $log$ in $Seq_{log}$ do
7:          if $log.\mathcal{S}_v = NULL$ then
8:              append $log$ to the $Seq$ with the $NONE$ key;
9:          else if $\exists(\mathcal{S}_v, Seq_{log})$ in $\mathcal{D}_{vl}$
10:             s.t. $log.\mathcal{S}_v \subseteq \mathcal{S}_v$ or $\mathcal{S}_v \subseteq log.\mathcal{S}_v$ then
11:             $\mathcal{S}_v \leftarrow \mathcal{S}_v \cup log.\mathcal{S}_v$;
12:             $Seq_{log}.append(log)$;
13:         else
14:             $\mathcal{D}_{vl} \leftarrow \mathcal{D}_{vl} \cup (log.\mathcal{S}_v, [log])$;
15:         end if
16:     end for
17:     if in training process then
18:         $\mathcal{D}_{ti} \leftarrow$ UpdateSubroutine($\mathcal{D}_{ti}$, $\mathcal{D}_{vl}$);
19:     end if
20: end for
```

Subroutine. The Intel Keys in an entity group are built as subroutines that are sequences of Intel Keys. Some subroutines can concurrently run multiple instances during execution distinguished by identifiers. Algorithm 2 builds Intel Keys in one entity group into subroutines. Before applying the algorithm, we find all log sequences belonging to the entity group generated by different executions. The set of log sequences is represented as $\mathcal{S} < Seq_{log} >$. The algorithm takes $\mathcal{S} < Seq_{log} >$ as the input to build subroutines. It maintains a dictionary $\mathcal{D}_{vl}$ that maps identifier types to Intel Key sequences. The dictionary $\mathcal{D}_{vl}$ is a temporary storage for each session that maps the actual identifiers to log message sequences. Before consuming a session, it clears $\mathcal{D}_{vl}$ and assigns an empty sequence with the key $NONE$ (Line 5). The empty sequence is used to store log messages that has no identifiers. In one session, it iterates each log message in the sequences (Line 6). It first updates the log sequence with the $NONE$ key if the log message does not have an identifier (Line 7 ~ 8). If the identifier values a log message that has already been in this session, it updates the log sequence with corresponding identifiers (Line 9 ~ 12). Otherwise, it creates a new key-value pair in $\mathcal{D}_{vl}$ and adds the log message to the new sequence (Line 14). After each session, it updates the subroutines based on the log sequence in this session.

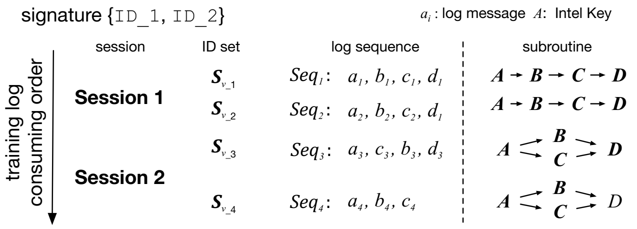

signature {ID_1, ID_2}
a_i : log message A: Intel Key
session
ID set
log sequence
subroutine
training log consuming order
Session 1
Session 2
S_{v_1}
S_{v_2}
S_{v_3}
S_{v_4}
Seq_1: a_1, b_1, c_1, d_1
Seq_2: a_2, b_2, c_2, d_1
Seq_3: a_3, c_3, b_3, d_3
Seq_4: a_4, b_4, c_4
A → B → C → D
A → B → C → D
A B C D
A B C D

Figure 5: Illustration of the UpdateSubroutine function.

The UpdateSubroutine function maintains an order relation set of Intel Keys. With a focus on building the order relations, we assign each identifier with a corresponding identifier type represented by a capitalized word. For example, '`container_01`' and '`container_02`' have a type of '`CONTAINER`'. For each session, it first groups the key-value pairs in $\mathcal{D}_{vl}$ according to the corresponding identifier types of the identifiers in $\mathcal{S}_v$. The set of identifier types can be considered as a signature. For a signature, it iterates through the $(\mathcal{S}_v, Seq_{log})$ pairs and checks the order of the corresponding Intel Keys of the log messages in Seq_{log}. If Key_1 always appears before Key_2 in every Seq_{log}, it assigns a BEFORE relation $Key_1 \longrightarrow Key_2$ to the two Intel Keys. Otherwise, these two Intel Keys can appear in parallel. The function also keeps track of the Intel Keys that always appear in a subroutine and marks them as critical Intel Keys. IntelLog uses critical Intel Keys for anomaly detection. Figure 5 illustrates the subroutine construction process for the signature {`ID_1`, `ID_2`}. It first consumes two log sequences in session 1 that have the same order of Intel Key sequences. IntelLog takes this key sequence as the subroutine. Every Intel Key in this sequence is marked as critical (bold font). Once IntelLog consumes Seq_3 in session 2, it finds that the log messages of B and C are interchangeable. As a result, it breaks the BEFORE relation between B and C and assigns them as parallel. Finally, in Seq_4 there is no log message matching Intel Key D. IntelLog marks Intel Key D as a not critical one (normal font).

HW-graph. The lifespan of an entity group in a session is defined by the duration between the first and the last log message that belong to the group. We construct the hierarchical relationships by checking the lifespan of each entity group. The idea is that if the lifespan of an group LS_a is always within the lifespan of another group LS_b in every session, group a is dependent on group b and should be a child of group b. In addition, two groups can either execute sequentially or in parallel. In order to capture the relationships, we define three relations as described in Figure 6. Two entity groups are assigned with `PARENT` or `BEFORE` only if they satisfy such relations in every log session. Otherwise, they execute in parallel and are assigned with `PARALLEL`.

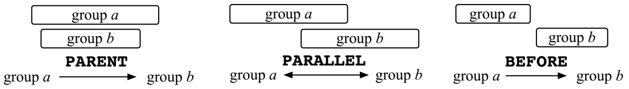
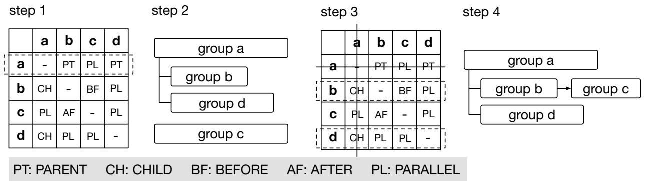


Figure 6: Three relations between entity groups.



step 1

	a	b	c	d
a	-	PT	PL	PT
b	CH	-	BF	PL
c	PL	AF	-	PL
d	CH	PL	PL	-

step 3

	~~a~~	b	c	d
~~a~~	~~-~~	~~PT~~	~~PL~~	~~PT~~
b	~~CH~~	-	BF	PL
c	~~PL~~	AF	-	PL
d	~~CH~~	PL	PL	-

Figure 7: The building procedure of the HW-graph based on entity relations.

IntelLog constructs the HW-graph of the targeted system based on the relations between each pair of entity group. We use an example to illustrate the construction procedure, as shown in Figure 7. For the simplicity of explanation, we add two auxiliary group relations `CHILD` and `AFTER` that are the opposites of `PARENT` and `BEFORE` respectively. First, IntelLog identifies the groups that only have `PARALLEL`, `PARENT` and `BEFORE` relations, say group *a* (step 1). Then, IntelLog constructs other groups based on their relations with group *a*. In this case, group *b* and *d* are children of group *a*, and group *c* executes with *a* in parallel (step 2). Once group *a* is built, IntelLog crosses out all the relations that are associated with group *a* (step 3). IntelLog repeats this procedure until all groups are crossed out. At this point, IntelLog constructs a complete HW-graph of a targeted system (step 4).

4.2 Anomaly Detection

When consuming incoming logs, IntelLog instantiates a HW-graph instance for each session of the targeted system. A HW-graph instance has the same entity group hierarchy as the corresponding HW-graph. In each entity group, however, it has multiple subroutine instances. For example, an entity group G in a HW-graph has a subroutine represented by a sequence of Intel Keys [A, B]. In the HW-graph instance, the entity group G may have two subroutine instances $[\mathrm{a}_1, \mathrm{b}_1]$ and $[\mathrm{a}_2, \mathrm{b}_2]$, where a_i and b_i are log messages. For each incoming log message, IntelLog uses Algorithm 2 to determine its subroutine instance. For an instance, if the corresponding Intel Key of an incoming log message is a critical one, IntelLog marks the critical Intel Key as used in the subroutine sequence.

IntelLog reports two kinds of anomalies: 1) unexpected log messages, and 2) erroneous HW-graph instances. These anomalies are common in a multi-tenant data cluster since tasks can be affected by administrators or other user processes.

Unexpected log message. IntelLog reports log messages that are not matched with any Intel Key. Furthermore, IntelLog tries to extract information of the five fields from the unexpected messages using the approaches of POS tagging and structure parsing. Evaluation shows that such information helps users diagnose the erroneous components and the root causes.

Erroneous HW-graph instance. When a whole session is consumed, IntelLog reports the erroneous HW-graph instances. Such instances either have an erroneous hierarchy of entity groups, abnormal subroutine instances or missed critical Intel Keys. IntelLog reports the problematic entity groups or subroutine instances.

5 IMPLEMENTATION

We implement IntelLog in about 6,700 lines of Java code and 200 lines of Python code. We have its package open-source at Github, `https://github.com/EddiePi/IntelLog/`. The code package also includes about a 400-line implementation of Spell. Spell defines a threshold t that helps matching log messages to log keys. We empirically set t to 1.7 via our experiments. We use OpenNLP [3] as the POS tagging tool and use Stanford Parser [12] to analyze the structures of log keys. Both HW-graphs and its instances are output as JSON files which can be queried by JSON query tools such as JSONQuery [2].

We deploy three popular distributed data analytics frameworks, Hadoop MapReduce, Spark and Tez as our targeted systems. All three systems are managed by Yarn [34], a cluster resource management framework. Execution in Yarn is encapsulated inside containers. Thus, we consider the log messages generated by one container as a log session. Since the formats of logs generated by different systems vary, we implement two log formatters for the targeted systems in about a total of 200 lines of Java code. The formatters simply recognize the log formats such as timestamps, output classes and log contents by pattern matching. Note that for new systems, users need to customize and implement their own formatters.

We omit log messages that only consist of a set of key-value pairs since they are not written in natural language. They can be captured by pattern matching In the training phase. We also use Spell to discover the log keys of these omitted log messages. IntelLog maintains a list of these log keys. Once a log message matches a log keys in the list, IntelLog ignores them instead of triggering the unexpected message errors.

6 EVALUATION

We evaluate IntelLog from three aspects. 1) We evaluate the accuracy of information extraction for Intel Keys. (2) We use a case study to illustrate the HW-graph that represents the workflow of a targeted system. We evaluate the effectiveness of HW-graph and compare it with the S^3 graph in Stitch [39]. 3) We evaluate the problem detection accuracy of IntelLog and compare it with related work DeepLog [18] and LogCluster [21]. We also demonstrate how HW-graphs can help users diagnose the root causes of anomalies.

6.1 Experiment Setup

We conduct the experiments on a 27-node physical cluster (1 master node and 26 worker nodes) managed by Yarn [34]. The cluster is connected with 10-Gbps Ethernet. Each node has 128 GB memory, 4 Intel Xeon E5-2640 CPUs (8 cores per CPU, 32 cores in total) and is installed with Ubuntu 16.04 with Linux kernel 4.12.8. The versions of the targeted data analytics systems are Spark-2.1.0, Hadoop MapReduce-2.9.1 and Tez-0.8.4, respectively.

We implement a workload generator that submits jobs for each targeted system. For Spark and MapReduce, the generator randomly

Table 4: Accuracy of information extraction by IntelLog in the three systems.

Frame- works	Consumed log msg.	# of Intel Keys	Entities (Total / FP / FN)	Identifiers (Total / FP / FN)	Values (Total / FP / FN)	Locations (Total / FP / FN)	Operations (Total / Missed)
Spark	1,361,008	60	63 / 3 / 0	19 / 1 / 1	13 / 1 / 0	9 / 0 / 1	63 / 5
MapReduce	5,254,050	44	43 / 9 / 2	11 / 1 / 1	41 / 1 / 1	1 / 0 / 0	45 / 5
Tez	1,127,549	95	101 / 2 / 3	13 / 0 / 3	43 / 3 / 0	3 / 0 / 0	97 / 7
Total	7,742,607	219	207 / 14 / 5	43 / 2 / 5	97 / 5 / 1	13 / 0 / 1	205 / 17

chooses jobs from HiBench[19] to generate the workloads. HiBench includes a wide range of jobs including text processing, machine learning and graph processing. For Tez, we use Hive-1.2.2 [32] as the query interface. The generator randomly chooses queries from TPC-H [7] as the workloads. TPC-H is a suite of database queries that have broad industry-wide relevance. In the model training phase, resource configurations are carefully tuned in the generator in order to guarantee successful and normal execution of every job. In § 6.2 and § 6.3, we use the generator to randomly submit 100 jobs to each system. The logs are collected for evaluation.

6.2 Information Extraction

In IntelLog, accuracy checking is done by comparing Intel Keys with log messages. However, such an approach has a possibility to incorrectly categorize an identifier as an value or vice versa. The reason is that such fields may only contain numeric values that can appear to be ambiguous even with the context. Since the source code of the targeted systems is available, we check the accuracy of information extraction by manually comparing Intel Keys with the corresponding logging statements in the source code.

Table 4 shows the accuracy of information extraction for each field. 'Total' denotes the manual checking results. 'FP' and 'FN' denote false positive and false negative, respectively. We present the total number and the missed number of operations, since there is no false positive operations (Other fields cannot be categorized as operations). For the entity field, the major reason of false positives is that IntelLog categorizes abbreviations as entities, such as 'ref' for 'reference' and 'tid' for 'task id'. We categorize such words as false positives since they are meaningless without context. False negatives are usually caused by entity phrases with four or more words. Note that IntelLog has relatively high accuracy for Tez. The reason is that most Tez logs are well formatted as a sentence followed by a set of key-value pairs. Thus, the entities can be correctly recognized. The false negatives of identifiers are also false positives of values. Such fields only contain numeric variables, which makes it difficult even for manual categorization. The only false negative of location is information about a service port categorized as a value by IntelLog.

For operation extraction, IntelLog performs well when analyzing sentences with correct grammatical structures. However, some log messages do not strictly follow grammatical rules. For instance, MapReduce outputs a log '`Down to the last merge-pass...`' in its critical path. This log message does not have a predicate so that IntelLog cannot recognize the operation. We observe two Tez log keys that record vague information even though they are grammatically correct. The sample log messages generated by the two keys are '`6 Close done`' and '`4 finished. Closing`'. These two log messages are actually related to query operators after we check the source code. Information extraction for such log messages is beyond the scope of this paper.

Table 5: Log and HW-graph statistics for the three systems.

Frame- works	length of sessions	# of groups all / crit	length of subroutines max / avg. all / avg. crit
Spark	347	45 / 10	10 / 1.2 / 2.3
MapReduce	137	35 / 13	19 / 1.7 / 2.8
Tez	304	59 / 27	14 / 2.7 / 4.6

6.3 HW-graph and Workflows

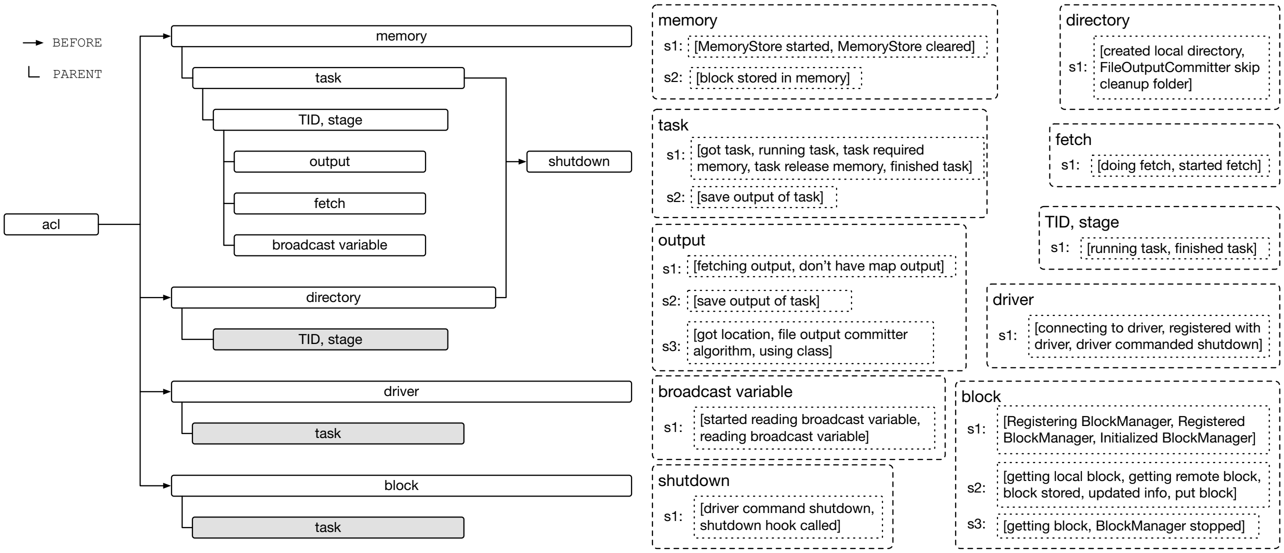
A HW-graph helps users to understand the workflows from two aspects. First, it categorizes workflows into entity groups in a hierarchical manner. This provides an overview of the system without inspecting the detail events in it. Second, it uses subroutines as a fine-grained view of workflows in each entity group, which only focuses on specific entities and filters out irrelevant ones. Subroutines are helpful when users try to understand parts of the workflow.

Since a HW-graph of a distributed data analytic system is large and contains rich information, we divide entity groups into two categories: critical groups and secondary groups. We categorize a critical group if it meets either of the following two criteria: 1) it contains multiple Intel Keys or, 2) it contains a Intel Key that has multiple corresponding log messages in a single session. The first criterion captures different entities that correlate with each other. The second criterion captures the entities that execute repeatedly or in parallel, which usually indicate common tasks. In practice, users can also choose to obtain a comprehensive HW-graph of a system that contains all the entity groups.

In order to evaluate how a hierarchical HW-graph can reduce the user efforts spent on understanding a workflow, we show the statistics of log sessions and the corresponding HW-graphs. Specifically, we measure the following metrics: 1) average number of log messages in a session; 2) number of all entity groups of a system; 3) number of critical entity groups of a system; 4) average length of a subroutine in all entity groups; and 5) average length of a subroutine in critical groups. Table 5 shows the results.

The number of entity groups (critical groups) are 5~10 (10~50) times fewer compared to the length of a session. Furthermore, the entity groups are organized in a hierarchical manner, which provides a clear view of workflows comparing to the original interleaved log messages. The length of a subroutine instance in entity groups are also shorter than that of a session. The longest instance has about 20 log messages, which is practical for manual analysis.

Spark HW-graph. Due to the limit of space, we solely illustrate the most complex HW-graph of Spark containing all the critical


BEFORE
PARENT
acl
memory
task
TID, stage
output
fetch
broadcast variable
shutdown
directory
TID, stage
driver
task
block
task
memory
s1: [MemoryStore started, MemoryStore cleared]
s2: [block stored in memory]
directory
s1: [created local directory, FileOutputCommitter skip cleanup folder]
task
s1: [got task, running task, task required memory, task release memory, finished task]
s2: [save output of task]
fetch
s1: [doing fetch, started fetch]
TID, stage
s1: [running task, finished task]
output
s1: [fetching output, don't have map output]
s2: [save output of task]
s3: [got location, file output committer algorithm, using class]
driver
s1: [connecting to driver, registered with driver, driver commanded shutdown]
broadcast variable
s1: [started reading broadcast variable, reading broadcast variable]
block
s1: [Registering BlockManager, Registered BlockManager, Initialized BlockManager]
s2: [getting local block, getting remote block, block stored, updated info, put block]
s3: [getting block, BlockManager stopped]
shutdown
s1: [driver command shutdown, shutdown hook called]

(a) Hierarchical relations between entity groups

(b) Subroutines in each entity group

Figure 8: The HW-graph for Spark containing the semantic knowledge of the workflow. We omit the children of the entity groups in gray rectangles since they have the same hierarchy as the children of the 'memory' group.

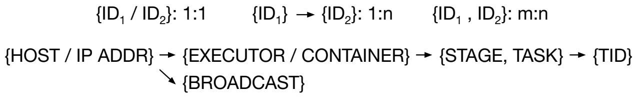


Figure 9: The S^3 graph of Spark built by Stitch.

groups. Figure 8 depicts the HW-graph that is built from over 1.2 million Spark log messages.

Figure 8(a) clearly illustrates the hierarchical relations between entity groups, which is a high-level view of Spark workflow: 1) Spark first checks the 'acl'; 2) Then, it has four major entities throughout execution, which are 'memory', 'directory', 'driver' and 'block'; 3) Under these four entities, there are child entities such as 'task' and 'fetch'; 4) Group 'shutdown' is after 'task' and 'directory'. The graph shows that it can execute with operations in groups 'memory', 'driver' and 'block' . The length of a rectangle indicates the lifespan of an entity group. We omit the children of the entity groups in grey rectangles since they are the same as those under group 'memory'.

Figure 8(b) shows the subroutines in each entity group and the Intel Keys in subroutines. For simplicity, Intel Keys are represented by the extracted operations. One advantage of using the entity group is that it categorizes correlated entities and the event of entities. We use group 'block' as an example to illustrate how subroutines in an entity group describe workflows. Group 'block' has three subroutines: 1) s1: a subroutine with identifiers of BlockManager; 2) s2: a subroutine with identifiers of block; and 3) s3: a subroutine with no identifier. Subroutine 1 records the workflow of a BlockManager that has three operations: 'registering', 'registered' and 'initialized'. Subroutine 2 records the blocks stored whether in memory or on disk during the execution. Subroutine 3 records the 'getting block' events with the number of blocks got. Note that the 'stopped' operation of a BlockManager is categorized into s3 since it has no identifier. The subroutines provide users with a clear view of the workflows that are related to the 'block' component.

Workflows of MapReduce and Tez can also be represented by HW-graphs. Similarly, entity groups capture related entities in the two systems. For example, group 'map' in MapReduce captures 'map metrics system' and 'map output'. Group 'task' in Tez captures 'task' and 'TaskAttempt'.

IntelLog vs. Stitch. Stitch [39] is a closely related tool that reconstructs the workflow of a targeted system as an S^3 graph based on identifiers. An S^3 graph defines four relationships between the identifier pairs in logs, i.e., empty, 1:1, 1:n, and m:n. The 1:1 relationship indicates that the two identifiers are interchangeable. The 1:n relationship indicates a hierarchical relationship between the two identifiers. The m:n relationship indicates that an object can only be unambiguously identified by the combination of the identifier pair. To compare the S^3 graph with the HW-graph, we reconstruct the S^3 graph of the Spark system.

Figure 9 shows that the S^3 graph reveals the hierarchical relationship between identifiers in Spark. For example, STAGE and TID have a 1:n relationship that indicates one stage runs multiple TIDs. However, a major limitation of S^3 is that it does not has semantic information recorded in logs. Users can only infer workflows from the names of identifiers, leaving a large amount of information unexploited. In practice, the reconstructed workflows by Stitch contains the lifespans of objects and their hierarchical relationships. On the other hand, IntelLog contains not only such kinds of information but also the operations and events related to objects. To this end, IntelLog is more fine-grained in terms of providing workflows to users because of its semantic awareness.

Table 6: The accuracy of anomaly detection by IntelLog.

Framework	number of sessions	length of a session	D / FP / FN / (P/B)
Spark	4∼26	20∼1812	13 / 2 / 2 / (2)
MapReduce	16∼257	67∼2147	15 / 1 / 0 (0)
Tez	2∼36	107∼486	13 / 3 / 2 (3)

6.4 Anomaly Detection

In order to evaluate the capability of anomaly detection by IntelLog, we develop a problem injection tool that emulates three real-world scenarios that were also studied in tools DeepLog [18] and LogCluster [21]. The problems include: 1) execution abortion of a session, 2) network failure on a node, and 3) node failure. The first problem can be caused by administrators or other user processes. We simulate it by sending a `SIGKILL` signal that gives the process no grace period to do cleanup. The second and third problems are common due to multiple reasons such as mis-configurations or hardware errors. We simulate network failures by disabling the network interface and simulate node failures by shutting down the machine. To generate logs with various lengths, we submit jobs to the cluster with five sets of configurations that have different input data sizes and resource allocations for each system. For each configuration setting, we submit three jobs injected with the three problems respectively, and submit three jobs that are not affected by the problems. Note that we tune the configurations such that these jobs are guaranteed to run successfully as when no problem is injected. The injection tool triggers the problem at a random point during the job execution. To summarize, we submit a total of 30 jobs for each system, 15 of which run with the problems.

Table 6 summarizes the anomaly detection results by IntelLog. For each system, we present the number of sessions and the length of one session. 'D' denotes the number of injected problems that are detected by IntelLog. 'FP' denotes the number of false positives and 'FN' denotes the number of false negatives. In experiments, we also find that IntelLog can capture performance problems and anomalies caused by bugs. The number of bugs is denoted as '(P/B)'. IntelLog detects 41 out of 45 injected problems. In addition, IntelLog captures 5 unexpected problems that are caused by inappropriate configurations or a internal bug. In general, IntelLog has a 87.23% precision and a 91.11% recall.

We manually check the system logs and the problem injection traces in order to analyze the causes of inaccurate anomaly detection. A typical reason of false positives is incomplete HW-graph due to insufficient training logs. We use Spark as an example to illustrate this scenario. During the shutdown phase, the Spark driver sends every Spark worker a '`shutdown`' command. Then, the driver itself enters the shutdown phase. The workers may still receive a heartbeat telling the disconnection of the driver and record it in the log file. Since the configurations are finely tuned in the training environments, workers finish the self cleanup procedure so quickly that they terminate before receiving the last heartbeat from the driver. In this case, the disconnected event does not show up in the logs. On the other hand, we change the resource configurations during the anomaly detection phase. The workers may experience a slower shutdown and output this log message. Since IntelLog does not capture this log message in the training phase, it categorizes this message as an unexpected one and alarms users with a potential anomaly concerning `shutdown`.

Table 7: Job descriptions in the three case studies.

Case No.	system / job name	input / resources	Session D / T	anomaly summary
1	MapReduce / WordCount	30GB / 8-core, 4GB	4 / 259	network problem on a host
2.1	Spark / KMeans	30GB 8-core, 2GB	1 / 8	a performance issue
2.2	Tez / Query 8	5GB 1-core, 1GB	24 / 25	a performance issue
3	Spark / WordCount	30GB 8-core, 16GB	4 /8	an internal bug of Spark

Next, we use three case studies to demonstrate how IntelLog reports anomaly execution and helps users to diagnose the root causes. The workload and anomaly information is shown in Table 7. In column 'sessions D / T', 'D' denotes the number of problematic sessions reported by IntelLog while 'T' denotes the total number of sessions of the job.

Injected problem. In the first case, we take advantage of IntelLog and detect that a MapReduce WordCount job experiences a network problem on a host. We show the step-by-step procedure that IntelLog leads us to the root cause. After consuming the logs output by this job, IntelLog reports four problematic sessions containing unexpected log messages out of a total of 259 sessions. At this point, IntelLog already significantly reduces the log range for analysis. Then, IntelLog applies the information extraction procedure on the unexpected log messages to transform them to Intel Messages. The result indicates that all of the unexpected messages belong to the '`fetcher`' entity group. We apply a `GroupBy` operator on the Intel Messages based on their identifiers. The result shows that 11 groups have log messages indicating host connection failures. Finally, we apply another `GroupBy` operator based on the location information. The result only has one group with a group ID 'host A'. In other words, the diagnosis procedure shows that 11 fetchers have a problem when connecting to 'host A' during execution. After checking the trace of the injection tool, we find that a network failure is injected during the job execution.

Performance issue. The second case illustrates that IntelLog has the capability to report potential performance issues even if jobs finish successfully. When IntelLog consumes logs from a Spark KMeans job and three Tez jobs (Query 8) without injected problems, it reports unexpected log messages that are caused by a performance problem after further inspection. IntelLog detects that one Spark session and 71 Tez sessions are problematic. IntelLog extracts a new entity '`spill`' from the unexpected log messages. Also, the unexpected logs from Tez record a file path of a disk location. In this case, the '`spill`' event stores the intermediate data to the disk when the memory usage reaches the configured limit, which incurs additional I/O overhead. Then, we re-run the two jobs with all the same configurations but a larger memory limit to verify whether the memory limit causes the anomaly. The resulted logs are consumed by IntelLog without triggering a problem.

Table 8: Comparison of the anomaly detection accuracy among IntelLog, DeepLog and LogCluster.

tools	precision	recall	F-measure
IntelLog	87.23%	91.11%	89.13%
DeepLog	8.81%	100.00%	16.19%
LogCluster	73.08%	N/A	N/A

An unexpected bug. In the last case, IntelLog automatically reports an unexpected anomaly of a Spark WordCount job which finally turns out to be a bug in Spark (Spark-19731 [6]). In this case, the Spark job successfully finishes and it does not generate any unexpected log messages. However, IntelLog reports that 4 out of 8 Spark sessions do not contain any log message of the '`task`' entity group nor of its child groups, which are shown in Figure 8. We further inspect the HW-graph instances built from the other 4 sessions. Each session has at most 8 subroutine instances in the task entity group, which indicates that a Spark container only gets assigned at most 8 tasks. Per a previous study LRTrace [28], each Spark container without a task occupies at least 250 MB memory resource but it does not contribute to the job progress. In this case, at least 1 GB memory is wasted in the cluster during the job execution. A possible solution is that Spark should take the input data size into consideration when it launches containers. Note that a concrete solution is beyond the scope of this paper.

Comparison Table 8 shows the anomaly detection accuracy of IntelLog, DeepLog [18] and LogCluster [21]. IntelLog achieves the best overall performance. Its precision and F-measure are significantly highly than those by the other two tools. LogCluster aims to reduce human efforts in manually examining logs. Though most of its reported logs are related to anomalies (high precision of 73.08%), it may still miss some logs caused by the problems (recall N/A). DeepLog has a high accuracy rate when it is applied to HDFS and OpenStack systems. However, its performance degrades when it targets distributed data analytics systems (low precision of 8.81%). The reason is that data analytics jobs have much higher parallelism than infrastructure-level distributed systems. For example, the length of a log sequence generated by a VM operation request in OpenStack is relatively small and fixed. In such a case, DeepLog can accurately predict the next log in a sequence by the logs seen before. But in data analytics systems, parallelism exists even in one session of a single job. For example, multiple tasks run in one Spark executor, and multiple fetchers run in one MapReduce container. Therefore, the log orders are less predictable. Another fundamental difference between these two kinds of systems is that the data size affects the length of log sessions of data analytics systems. The log length of new job sessions is difficult to be predicted from the job history.

Summary. The experiments show that IntelLog is able to detect anomalies of the systems, including simulated real-world problems, performance issues and internal system bugs. For the reported anomalies including the three cases above, IntelLog also pinpoints the problematic entity groups or subroutines. By using such information, users can easily locate the root causes. In addition, Intel Messages represented by key-value pairs can be stored as JSON files or in database, which facilitates log analysis since users can use queries to obtain related messages of problems.

7 DISCUSSIONS

N-gram extraction [35] is a conventional approach for pattern extraction. However, the approach is not efficient for the information extraction process in IntelLog for two reasons. It is time consuming to perform the N-gram approach since it searches all possible sequences that has a length of N. The other reason is that it may capture word sequences that are not entities such as 'for attempt', which results in a larger result set with many useless phrases.

The accuracy of information extraction depends on the quality of logs. IntelLog achieves a relatively higher accuracy in Tez compared to in MapReduce and Spark because logs of Tez are usually short and well formatted. Factors that affect the accuracy include using multiple phrases to describe the same entity and using the same phrase to indicate multiple entities.

The structure parsing based extraction approach of IntelLog is effective because logs are usually written in simple sentences that have only one clause. For complex sentences, the approach may miss the operations in the dependent clause but it can still extract the operations in the independent clauses.

One limitation of IntelLog is that it only focuses on logs that are written in natural languages. Otherwise, the HW-graphs may not be constructed due to the lack of correlated entity names. Fortunately, popular distributed systems use natural languages when generating log message.

8 RELATED WORK

Log analysis. Many previous studies focus on log analysis [14, 15, 18, 26, 28, 29, 36, 38, 39]. Workflow reconstruction from logs receives much attention. Stitch [39] uses an identifier-based approach to construct workflows of systems, leaving the rest of the information unused. Furthermore, identifiers are usually abbreviations, which make it difficult for users to understand the systems. A fundamental drawback is that it requires manual analysis to detect the anomalies, which is only practical with small-sized jobs (e.g., jobs with fewer than 50 sessions). CloudSeer [36] uses an automation based approach to reconstruct the workflow of a cloud infrastructure. Since the length and order of the log sequences generated by the cloud infrastructure are relatively fixed, it can accurately capture the workflow and detect anomalies. However, it cannot be applied to distributed data analytics systems since the lengths and orders of logs in such systems can vary significantly.

There are studies that leverage static analysis to conduct log analysis [38]. They require the byte code of the systems and need to perform the static analysis once the systems are upgraded. In addition, it requires user efforts and has limited capability for specific problems. There are also many studies applying machine learning methods on log analysis [9, 15, 26]. PerfScope [15] applies a clustering algorithm on OS logs to find logs that have similar behaviors. DeepLog [18] is a recent work in this category. It uses LSTM, a deep neural network, to obtain the probability of the next log in a log sequence. For machine learning approaches, the training accuracy relies on 1) the quality of the training data, and 2) the parameters in the machine learning models. NetSieve [30] uses an NLP based approach to extract semantic information from network tickets. A fundamental difference between NetSieve and IntelLog is that NetSieve focuses on logs hand-written by network administrators.

Such logs only record maintenance information but do not have workflow information for reconstruction.

Different from those tools, IntelLog extracts the semantic knowledge in logs and leverages it to reconstruct the workflows of the targeted data analytics systems. The semantic knowledge is essential for users to understand the systems. When an anomaly occurs, IntelLog not only infers the problematic requests but also the potential erroneous components, which significantly helps users to locate the root causes.

Intrusive approach. This is a popular approach to obtain information from inside targeted systems [13, 23, 25], which requires source code modification. The obtained information are guaranteed to be accurate an0d unambiguous. However, an intrusive approach needs to balance the trade-off between the amount of information and overhead. Dynamic intrusive tools such as Pivot Tracing [23] insert tracing points into the target systems on the fly, which reduce the overhead. However, users need to know the intended function and metric names in the source code. A recent study AUDIT [22] adopts both log analysis and the intrusive approach. It sets triggers in the targeted systems and inserts more logging statements in the targeted systems when triggers are fired by anomalies. In this way, the additional logging statements only introduces overhead when the systems need maintenance because of anomalies.

Fundamentally different to intrusive approaches, IntelLog is non-intrusive. It does not require modification or access to the source code of the targeted systems. A non-intrusive approach is more ubiquitous since most systems output logs while fewer systems make their source code available.

9 CONCLUSION AND FUTURE WORK

This paper presents IntelLog, a workflow reconstruction and anomaly detection tool for distributed data analytics systems. It reconstructs the workflows of the systems by using NLP based approaches in a non-intrusive manner. It uses POS analysis and sentence structure parsing to extract information from log keys. The results are Intel Keys, an enhanced representation of the original log keys. IntelLog leverages the nomenclature of entity names to group together related entities. Its semantic awareness by HW-graphs allows users to easily understand the targeted systems. IntelLog also uses the HW-graphs for anomaly detection. Experimental results show that IntelLog outperforms existing tools in automatically detecting anomalies caused by real-world problems, performance issues and internal system bugs and pinpoints the erroneous components. Logs in natural languages help users not only analyze anomalies but also understand the underlying mechanism of the systems.

In the future, we plan to extend IntelLog to distributed machine learning systems (e.g., TensorFlow).

Acknowledgment

This research was supported in part by U.S. NSF awards SHF-1816850 and CNS-1422119.

REFERENCES

[1] Graphite. https://graphite.readthedocs.io/.
[2] JSONQuery. https://github.com/burt202/jsonquery/.
[3] OpenNLP. https://opennlp.apache.org/, .
[4] OpenStack. https://www.openstack.org/, .
[5] OpenTSDB. http://opentsdb.net//, .
[6] Spark-19371. https://issues.apache.org/jira/browse/SPARK-19371/.
[7] TPC-H. http://www.tpc.org/tpch/.
[8] TensorFlow. https://www.tensorflow.org/.
[9] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proc. of ACM SIGSOFT ESEC/FSE*, 2011.
[10] D. Borthakur. Hdfs architecture guide. hadoop apache project, 2008.
[11] Brid, Steven, E. Loper, and E. Klein. *Natural Language Processing with Python.* O’Reilly Media Inc., 2009.
[12] D. Chen and C. D. Manning. A fast and accurate dependency parser using neural networks. In *Proc. of ACL EMNLP*, 2014.
[13] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *Proc. of USENIX ATC*, 2017.
[14] W. Chen, A. Pi, S. Wang, and X. Zhou. Characterizing scheduling delay for low-latency data analytic workloads. In *Proc. of IEEE IPDPS*, 2018.
[15] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, Nipun, Arora, and G. Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proc. of ACM SoCC*, 2014.
[16] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. of ACM Communications*, 2008.
[17] M. Du and F. Li. Spell: Streaming parsing of system event logs. In *Proc. of IEEE ICDM*, 2017.
[18] M. Du, F. Li, G. Zheng, and V. Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proc. of ACM CCS*, 2017.
[19] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The HiBench benchmark suite: Characterization of the mapreduce-based data analysis. In *Proc. of IEEE Data Engineering Workshops (ICDEW)*, 2010.
[20] J. S. Justeson and S. M. Katz. Technical terminology: some linguistic properties and an algorithm for identification in text. *Natural Language Engineering*, 1995.
[21] Q. Lin, H. Zhang, J.-G. Lou, Y. Zhang, and X. Chen. Log clustering based problem identification for online service systems. In *Proc. of IEEE/ACM ICSE*, 2016.
[22] L. Luo, S. Nath, L. R. Sivalingam, M. Musuvathi, and L. Ceze. Troubleshooting, transiently-recurring problems in production systems with blame-proportional logging. In *Proc. of USENIX ATC*, 2018.
[23] J. Mace, R. Roelke, and R. Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proc. of ACM SOSP*, 2015.
[24] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, June 1993. ISSN 0891-2017.
[25] M. Mejbah ul Alam, T. Liu, G. Zeng, and A. Muzahid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proc. of ACM Eurosys*, 2017.
[26] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. of USENIX NSDI*, 2012.
[27] J. Nivre, M.-C. Marneffe, F. Ginter, Y. Goldberg, J. Hajic, C. D. Manning, R. McDonald, S. Petrov, S. Pyysalo, N. Silveira, R. Tsarfaty, and D. Zeman. Universal dependencies v1: A multilingual treebank collection. In *Proc. of LREC*, 2016.
[28] A. Pi, W. Chen, X. Zhou, and M. Ji. Profiling distributed systems in lightweight virtualized environments with logs and resource metrics. In *Proc. of ACM HPDC*, 2018.
[29] A. Pi, W. Chen, W. Zeller, and X. Zhou. It can understand the logs, literally. In *Proc. of IPDPSW*, 2019.
[30] R. Potharaju, N. Jain, and C. Nita-Rotaru. Juggling the jigsaw: Towards automated problem inference from network trouble tickets. In *Proc. of USENIX NSDI*, 2013.
[31] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proc. of ACM SIGMOD*, 2015.
[32] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. of VLDB Endowment*, 2009.
[33] K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proc. of HLT-NAACL*, 2003.
[34] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache Hadoop YARN: Yet another resource negotiator. In *Proc. of ACM SoCC*, 2013.
[35] M. Yamamoto and K. W. Church. Using suffix arrays to compute term frequency and document frequency for all substrings in a corpus. *Computational Linguistics*, 27(1):1–30, Mar. 2001. ISSN 0891-2017.
[36] X. Yu, P. Joshi, J. Xu, and G. Jin. CloudSeer: Workflow monitoring of cloud infrastructures via interleaved logs. In *Proc. of ACM ASPLOS*, 2016.
[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. of USENIX HOTCLOUD*, 2010.
[38] X. Zhao, Y. Zhang, D. Lion, M. FaizanUllah, Y. Luo, D. Yuan, and M. Stumm. Iprof: A non-intrusive request flow profiler for distributed systems. In *Proc. of USENIX OSDI*, 2014.
[39] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proc. of USENIX OSDI*, 2016.