Improving State-of-the-art Compression Techniques for Log Management Tools

Kundi Yao, Mohammed Sayagh, Weiyi Shang, and Ahmed E. Hassan

Abstract—Log data records important runtime information about the running of a software system for different purposes including performance assurance, capacity planning, and anomaly detection. Log management tools such as ELK Stack and Splunk are widely adopted to manage and leverage log data in order to assist DevOps in real-time log analytics and decision making. To enable fast queries and to save storage space, such tools split log data into small blocks (e.g., 16KB), then index and compress each block separately. Previous log compression studies focus on improving the compression of either large-sized log files or log streams, without considering improving the compression of small log blocks (the actual compression need by modern log management tools). The evaluation of four state-of-the-art compression approaches (e.g., *Logzip*, a variation of *Logzip* by pre-extracting log templates named *Logzip-E*, *LogArchive* and *Cowic*) indicates that these approaches do not perform well on small log blocks. In fact, the compressed blocks that are preprocessed using *Logzip*, *Logzip-E*, *LogArchive* or *Cowic* are even larger (on median 1.3 times, 1.5 times, 0.2 times or 6.6 times) than the compressed blocks without any preprocessing. Hence, we propose an approach named *LogBlock* to preprocess small log blocks before compressing them with a general compressor such as *gzip*, *deflate* and *lz4*, which are widely adopted by log management tools. *LogBlock* reduces the repetitiveness of logs by preprocessing the log headers and rearranging the log content leading to an improved compression ratio for a log file. Our evaluation on 16 log files shows that, for 16KB to 128KB block sizes, the compressed blocks by *LogBlock* are on median 5% to 21% smaller than the same compressed blocks without preprocessing (outperforming the state-of-the-art compression approaches). *LogBlock* achieves both a higher compression ratio (a median of 1.7 to 8.4 times, 1.9 to 10.0 times, 1.3 to 1.9 times and 6.2 to 11.4 times) and a faster compression speed (a median of 30.8 to 49.7 times, 42.6 to 53.8 times, 4.5 to 6.0 times and 2.5 to 4.0 times) than *Logzip*, *Logzip-E*, *LogArchive* and *Cowic*. *LogBlock* can help improve the storage efficiency of log management tools.

Index Terms—Software log compression, Software logging, Log management tools.

1 Introduction

Log data is generated by logging statements that developers place into the source code to record run-time information [1, 2]. Such information enables software operators to understand the field operations of a software system. Prior research leverages log data to support performance assurance [3, 4], capacity planning [5, 6], anomaly detection [7, 8] and system failures diagnosis [9]. Such log data is commonly preserved for a long period of time for historical analysis and for legal compliance [10].

Large-scale software systems generate a large amount of log data every day. Such overwhelming sizes of log data make it difficult to analyze. Therefore, log management tools are used to process, archive and analyze log data. The usage of such tools to streamline and optimize DevOps and quality assurance in the field is showcased in prior research [11, 12, 13].

Due to its massive size, log data is often compressed for storage. Compared to traditional log archiving that compresses the whole log file, log management tools split log data into small blocks to achieve higher performance in processing and querying log messages. Log management tools aggregate log information to support log retention, iterative log analysis and visualization. For example, popular log management tools such as ELK Stack [14] and Splunk [15] split log data into blocks of size 16KB and 128KB (by default) respectively [16, 17] (as shown in the examples of Table 1). Log blocks are then indexed before being separately compressed. Finally, for querying information from logs, log management tools decompress only the appropriate blocks. However, the cost of log management services is enormous. For example, Splunk charges $1,800 per year for each ingested Gigabyte of log data [18]. Therefore, an efficient approach to storing small log blocks is essential for reducing the cost of log management and storage.

- *Kundi Yao, Mohammed Sayagh, and Ahmed E. Hassan are with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada.*
  E-mail: {kundi, msayagh, ahmed}@cs.queensu.ca
- *Weiyi Shang is with the Department of Computer Science and Software Engineering, Concordia University, Canada.*
  E-mail: shang@encs.concordia.ca

TABLE 1: The block sizes that are used by popular log management tools.

Log Management Tool	Block size
ELK Stack [14] [19]	16KB/48KB/60KB
Splunk [15]	128KB
Sumo Logic [20]	64KB
Syslog Ng [21]	64KB
Nginx [22]	64KB
Rsyslog [23]	8KB
DataDog [24]	256KB
Sentry [25]	1000 characters

* The sources of the information about the block sizes are attached in the replication package[1].

While log management tools compress small log blocks, prior research efforts on log compression focus on the compression of large files. General compressors, such as *gzip*, *lz4* and *deflate*, are designed for compressing general data, which have different characteristics [26]. On top of these general compressors, several approaches have been proposed to preprocess log files in a way that they can better benefit from existing general compressors [27, 28, 29, 30, 31].

Recently, Liu et al. [27] proposed *Logzip*, a log preprocessing approach that regroups similar log lines into groups, with each group summarized by a single log line template, and each line is summarized using its associated template and the varying parameters that distinguish the line from the rest of the lines in its group. Similar approaches, such as *LogArchive* [32] that groups similar log data into clusters or *Cowic* [31] that compresses log data by columns with pre-trained compression models, are also designed by the intuition of identifying repetitive information from *large* log data. **To date, no studies have ever investigated improving the compression of small blocks of log data (even though this is the actual compression need by modern log management tools).**

Our preliminary evaluation (c.f. Section 4) shows that state-of-the-art log compression approaches (i.e., *Logzip*, a variation of *Logzip* by pre-extracting log templates named *Logzip-E*, *LogArchive* and *Cowic*) do not perform well on small-sized log blocks. The actual compression ratios of these approaches are even worse than general compressors like *gzip*. On median, the sizes of the compressed blocks that are preprocessed using *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* are 2.3 times, 2.5 times, 1.2 times and 7.6 times of the sizes of the same blocks when they are compressed *without* any preprocessing. Such compression approaches usually look for patterns in log data. However, small log blocks often do not have repetitive patterns so the pattern matching approaches will end up introducing overhead to the compressed data. Therefore, to improve the compression of small log blocks (the actual need by modern log management tools), we propose a preprocessing approach named *LogBlock* based on the characteristics of small log blocks.

LogBlock consists of two main steps which are complemented with four preprocessing heuristics: *Extract identical tokens*, *Delta encoding for numbers*, *Build dictionary for repetitive tokens* and *Extract a common prefix string*. Each of the heuristics aims to replace long or repetitive tokens in log data with shorter representations. We compare *LogBlock* against general compressors and the state-of-the-art log preprocessing approaches. Our evaluation results show that for the sizes of log blocks that are typically used by log management tools (16KB used by ELK Stack and 128KB used by Splunk), *LogBlock* improves the compression ratio of small blocks by a median of 5%, 9%, 15% and 21% for the blocks whose sizes are 16KB, 32KB, 64KB, and 128KB, respectively. *Logzip* does not outperform our approach at any block whose size is lower than 256KB. Examining *LogBlock*'s preprocessing heuristics, we observe that they increase the compression ratio as much as 6.4%, 48.5%, 22.2% and 9.7% for specific log files. Practitioners can optimize the performance of *LogBlock* by customizing the preprocessing heuristics according to their specific log files.

1. Our replication package is available at: https://queensuca-my.sharepoint.com/:f:/g/personal/18ky10_queensu_ca/EqAMNUbIxBRMqFoo9qHytvgBAenixikDvSqRXxo0O8EjrA?e=53Eg0U. We will open the access to this package on a GitHub repository when the paper is accepted.

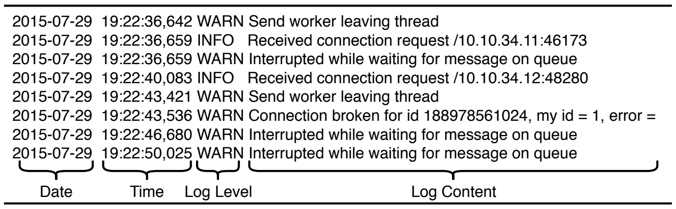
In summary, our paper makes the following contributions:

- ⋆ We highlight that current log compression efforts do not satisfy the needs (aka the usage scenarios) of log management tools.
- ⋆ We propose and evaluate a new approach (*LogBlock*) that outperforms general compressors and stat-of-the-art log compression approaches.

Paper structure: Section 2 summarizes the background and discusses the related work on improving log compression and retrieving information from compressed logs. Section 3 introduces the setup of the experiments in this paper. Section 4 evaluates the compression performance of the state-of-the-art log compression approaches on small log blocks. Section 5 explains the design of *LogBlock*. Section 6 evaluates the compression performance of *LogBlock* on small log blocks. Section 7 compares the compression performance achieved by the state-of-the-art log compression approaches and *LogBlock* on different block sizes. Section 8 discusses the threats to the validity of our findings. Finally, Section 9 concludes the paper.

2 Background and Related Work

Log lines are printed under pre-defined logging configurations. Thus, many log lines share a similar structure. As shown in Figure 1, each log line consists of fixed headers (date, time and log level) and a free-form component (log content) that is defined by developers.



```
2015-07-29  19:22:36,642 WARN Send worker leaving thread
2015-07-29  19:22:36,659 INFO  Received connection request /10.10.34.11:46173
2015-07-29  19:22:36,659 WARN Interrupted while waiting for message on queue
2015-07-29  19:22:40,083 INFO  Received connection request /10.10.34.12:48280
2015-07-29  19:22:43,421 WARN Send worker leaving thread
2015-07-29  19:22:43,536 WARN Connection broken for id 188978561024, my id = 1, error =
2015-07-29  19:22:46,680 WARN Interrupted while waiting for message on queue
2015-07-29  19:22:50,025 WARN Interrupted while waiting for message on queue
```

Fig. 1: A log snippet.

To support queries on log files, modern log management tools splits log files into small blocks then compress them. For instance, large-sized log files are split into small blocks, indexed, then compressed separately by a general compressor such as *gzip*. Thus, when querying a piece of log information, log management tools, like ELK Stack [14] and Splunk [15], only decompress the corresponding log blocks according to the indexes. For example, ELK Stack splits data into 16KB/60KB [33] (or 16KB/48KB [19]) blocks then compresses each block separately, while Splunk divides data into 128KB blocks.

Otten et al. [28] introduced the following common usage scenarios of log compression where small log blocks are needed:

- *Collecting log data to a central location for analysis*: only selected and compressed log data in small blocks are collected to minimize bandwidth usages.

- *Monitoring in real-time*: small log blocks are compressed to minimize data transfer time in order to support real-time monitoring.
- *Storing the log data compressed and later decompressing it for analysis*: log data is stored and compressed in small blocks for fast decompression before analysis, without the need for decompressing the entire log data.

In these scenarios, small log blocks are used for performance reasons. Compared to traditional log compression approaches, log management tools improve query efficiency by splitting a log file into blocks and only decompressing the needed blocks on demand. Thus, improving the compression ratio of small log blocks is critical.

Unfortunately, prior research on compressing log files [27, 28, 29, 30] does not study compressor performance in the context of the above common log compression scenarios (where the compression performance of small log blocks is critical).

2.1 Compressing large log files

While several approaches [27, 28, 29, 30, 32, 34, 35, 36] are proposed to preprocess and compress log files, they focus on large files whereas modern log management tools compress small blocks. We summarize the prior log compression approaches into two categories, namely log transformation and text replacement.

2.1.1 Log Transformation.

A first line of research considers transforming existing log lines in a way to improve the size of the compressed logs. This line of research leverages two main approaches for such a transformation, namely log clustering and log transposing.
Log Clustering. Christensen et al. [32] cluster log lines into different buckets according to their textual similarity, then compress each bucket separately. Feng et al. [30] extract and compress the dynamic variants of the log lines within each bucket to further improve the compression ratio of a log file.
Log Transposing. Mell et al. [35] consider the log data as a matrix where each log line is a row in the matrix, then the matrix is transposed so that similar tokens are placed closer to each other. This preprocessing approach achieves a 21% improvement in the compression ratio of a log file.

2.1.2 Text Replacement.

Prior research replaces long and repetitive text in log files with shorter representations. In particular, different text replacement approaches are used based on the local repetitiveness (delta encoding) and the global repetitiveness (token and template replacement) of log data.
Delta Encoding. Balakrishnan et al. [29] propose a log preprocessing approach that encodes the different characters between adjacent lines. Their approach improves the compression ratio of supercomputer logs by 28.3%.
Token Replacement. Otten et al. [28] transform all timestamps and IP addresses in a log file to binary representations, then replace the static tokens in log files (i.e., static words and phrases) with shorter representations. Their approach improves the compression ratio by 32%. Skibiński et al. [34] extract the differences between adjacent log lines, use a dictionary to replace global repetitive tokens (e.g., dates, time, IP address). Their approach achieves 19.8% to 36.6% improvement in the compression ratio.
Template Replacement. Log templates are the static information of log content. Such templates are usually highly repetitive in log files. For instance, Hätönen et al. [36] find that the most frequent patterns (i.e., log templates) in firewall or application logs cover more than 95.8% of the log lines. Recently, Liu et al. [27] propose a log preprocessing approach (i.e., *Logzip*) that extracts log templates from log data and replaces each template with a shorter representation (e.g., a unique ID). *Logzip* improves the compression ratio by 1.3 to 15.1 times compared to the compression without any preprocessing. *Logzip* also outperforms two state-of-the-art log compression approaches (i.e., Cowic [31] and LogArchive [32]) in terms of compression ratio.

Our paper is different from the aforementioned research works since we focus on small blocks (the most common need for log management tools) rather than large log files.

2.2 Querying from compressed log data

Querying compressed log data is resource-intensive since the compressed data needs to be decompressed [31]. Therefore, to facilitate queries and to reduce unnecessary resource usage (for decompression), several approaches [31, 37, 38] have been proposed for querying compressed log data. Surti et al. [37] propose LittleLog, a general-purpose approach that supports log compression and querying. This approach is implemented based on Succinct [39], a distributed data store that enables queries on compressed data without storing indexes. Their query speed is 97% faster than *grep* occurrence count and 65% faster than *grep* line count. However, their compression ratio is much lower than general compressors like *gzip* and *bzip2*. Aceto et al. [38] introduce a lossy compression approach that transfers log lines into numeric matrices. Their algorithm has a compression ratio that is close to *bzip2*. Recently, Lin et al. [31] propose an approach (i.e., *Cowic*) that separates log lines into columns (e.g. timestamps, IP addresses) and processes each column with different models, each of which is trained from a fraction of log dataset. *Cowic* has a similar compression ratio as *gzip*, but it is 3.6 to 71.1 times faster in query time when data is in memory, and 30.4% to 246.8% faster when data is on disk.

The aforementioned approaches focus on the fast querying of logs while having a similar or lower compression ratio than general compressors. We propose an approach that achieves a higher compression ratio on small blocks of log data while supporting the querying and analytics mechanisms of state-of-the-art log management tools.

2.3 Software logging in binary format

Developers usually record logs through off-the-shelf logging libraries such as Log4J2 [40] and LogBack [41]. A major cost for using such libraries for logging in practice is the performance overhead that relates to the execution time and resource consumption [42]. To minimize the logging overhead, Yang et al. [43] propose NanoLog, a logging system with low latency and high throughput. NanoLog extracts static logging information at compilation phase and only logs dynamic information in a binary format at runtime to save

TABLE 2: Log files that are obtained from a publicly available benchmark (i.e., Loghub) [45].

System Type	System Name	Description	Log Size
Standalone application	Proxifier	Log data collected from Proxifier client	2.5MB
Server applications	Apache	Error logs collected from Apache web server	4.9MB
	OpenSSH	Log data collected from OpenSSH server	71MB
Mobile systems	Andriod	Log data collected from an instrumented Android device	184MB
	HealthApp	Log data from the Health app on Android device	23MB
Operating systems	Windows	An aggregated collection of Windows event logs	27GB
	Linux	System log collected from Linux system	2.3MB
	Mac	System log collected from Mac OS	17MB
Super-computers	BGL	A publicly available dataset from BlueGene/L supercomputer system	709MB
	HPC	Log data collected from high performance computing cluster	32MB
	Thunderbird	Log data collected from the Thunderbird supercomputer system	30GB
	HDFS	Log data generated from Hadoop Distributed File System	1.5GB
	Hadoop	Log data from Hadoop Mapreduce with deployment failures	47MB
Distributed systems	Spark	An aggregated collection of Spark system logs	2.8GB
	Zookeeper	An aggregated collection of Zookeeper system logs	10MB
	OpenStack	Log data from OpenStack VMs with both normal and abnormal logs	59MB

I/O bandwidth. The static and binary information is later on recombined to generate human-readable log messages post execution. Marron et al. [44] introduce Log++, a logging system for JavaScript applications that copies semantic logging formats then compresses and converts dynamic information in an in-memory buffer during runtime.

The aforementioned approaches operate on the source code to reduce the logging overhead. Although such approaches are shown to be effective, log management tools process logs for applications for which they do not have access to the source code nor the ability to directly manipulate it. On the other hand, our paper proposes a log compression approach that can be directly adopted by log management tools without the need of manipulating the source code of the application that generated the logs.

3 Experimental Setup

In this section, we present the setup of our experiments, including the selection of our studied log data and compressors, together with our experimental environment.

3.1 Selected log files and compressors

Our evaluation considers 16 log files from a recent benchmark for log parsing [46]. These files range in size from 2.5MB to 30GB. The selected log files cover different types of systems, including distributed systems, mobile systems, operating systems, supercomputers, standalone applications, and server applications, as detailed in Table 2. In addition, we use the *gzip*, *lz4* and *deflate* compressors, which are widely adopted by log management tools [16, 17], to evaluate the compression of log files.

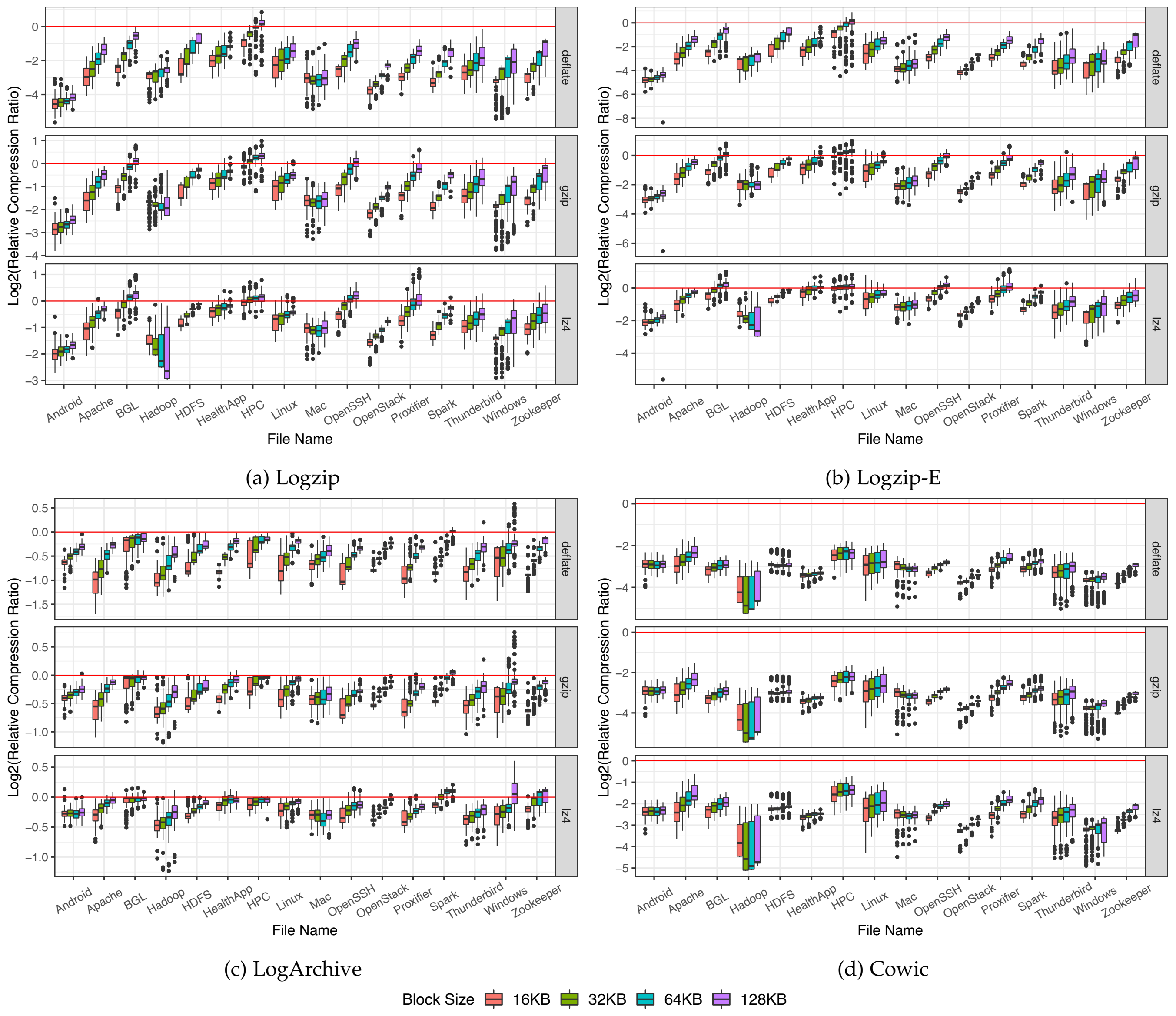
3.2 Experimental environment

We run our experiments on an Ubuntu v18.04 server with a 4-core Intel Core i5-4690 @ 3.50GHz CPU and 32GB memory. This server is mounted with an SSD storage of 1.8TB. We adopt the mentioned compressors in Section 3.1 to evaluate the compression performance of each file in terms of compression ratio and compression speed. We disable other running processes on the server to avoid any influences due to resource contention. We capture the preprocessing and compression times for each file.

4 Preliminary Study: The performance of the state-of-the-art compression approaches on small log blocks.

Motivation: In this section, we investigate the performance of *Logzip*, *LogArchive* and *Cowic* on the compression of small-sized log blocks. Prior log compression approaches such as *Logzip* [27], *LogArchive* [32] and *Cowic* [31] preprocess log files before compressing them. Such approaches improve the compression of large-sized log files or log streams. However, the performance of these approaches on compressing small log blocks, which are used by log management tools, has never been examined. *Logzip* compresses log files by extracting repetitive templates and variables from the log content and replacing them with indexes. *LogArchive* is a bucketing based compression approach, which groups similar log data into buckets according to their content similarity then compresses each bucket with a general compressor. Finally, *Cowic* is a model-based compression approach for streaming logs that compresses log entries using pre-trained models. These three aforementioned approaches depend heavily on the global repetitiveness of log data, while such high repetitiveness may not occur in small log blocks. In addition, we evaluate the compression performance of a variation of *Logzip*, which we refer to as *Logzip-E*. *Logzip-E* first extracts log templates from the original log file before it is split into small blocks. Then, those templates are used when compressing the blocks of log files. In other words, we introduce an extra template extraction step in *Logzip* that captures the global repetitiveness from the original log file. The extracted templates are then stored and reused in the future. The intuition behind *Logzip-E* is that the extracted templates from small log blocks may not be accurate due to their small sizes. The inaccurate templates may lead to a low performance of *Logzip*, which can be further improved if one has accurate templates from the logs. Therefore, *Logzip-E* leverages the whole log file to extract templates, which are later directly applied on each small log block.
Approach: To evaluate the performance of *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* on small blocks of a log file, we followed the approach shown in Figure 3. *Logzip-E* first extracts templates from the original log file then applies the templates to match log lines in every log block. The added template extraction step on the original log file ensures that we are leveraging the global repetitiveness in a file. The templates and their corresponding indexes are then stored in a separate file for future usages. Before we begin to preprocess log blocks, the template file is loaded for template matching in each log block.

For each size (16KB, 32KB, 64KB, and 128KB), we randomly extract 100 log blocks of the same size from each log file. We compare compressing each of the extracted blocks without any preprocessing against compressing the same block by preprocessing it using *Logzip*, *Logzip-E*, *LogArchive* or *Cowic*. When evaluating the performance of *Logzip-E*, we do not include the parsing time and the pre-loading time of the global templates since one may only need those



(a) Logzip

(b) Logzip-E

(c) LogArchive

(d) Cowic

Fig. 2: The relative compression ratio (in log-scale) between compression without and with preprocessing using *Logzip*, *Logzip-E*, *LogArchive* or *Cowic*. Each box shows the distribution of relative compression ratio of 100 blocks from the same log file using a specific block size (16KB, 32KB, 64KB and 128KB) and compressor (*gzip*, *lz4* and *deflate*). The horizontal red line at zero ($\log_2 1$) is the baseline that indicates the relative compression ratio without preprocessing. The actual compression performance values can be found in Table 5 and Table 6.

steps once for each type of log files. *Cowic* pre-trains a compression model with a portion of the log data [31]. When evaluating the performance of *Cowic*, we use the whole log block to train the compression model to avoid biased results. We use the relative compression ratio for such a comparison, as shown in Figure 3. The relative compression ratio is the compression ratio of the preprocessed block over the compression ratio of the original block. A relative compression ratio higher than one indicates that the log preprocessing approach has a better compression ratio than compression without any preprocessing. Furthermore, the higher the relative compression ratio for a log block is, the better that block benefits from the preprocessing step. We also perform a Wilcoxon rank-sum test [47] ($\alpha = 0.05$) to examine whether the compression performance is statistically different when leveraging the global and local repetitiveness using *Logzip-E* and *Logzip*, respectively.

Our evaluation considers sizes as well as the compressors that are used by modern log management tools. From our initial investigation, we find that log management tools usually set the block sizes between 16KB to 128KB. In addition, dictionary-based compressors (e.g., *lz4*, *deflate*, and *gzip*) are adopted by these tools. Therefore, we evaluate the compression performance of the log preprocessing approaches using three different compressors: *lz4*, *deflate*, and *gzip* on log blocks of size: 16KB, 32KB, 64KB, and 128KB.

To further understand the performance of log preprocessing approaches on log blocks, we perform a Spearman correlation analysis to investigate the association between the characteristics of log blocks (i.e., number of templates

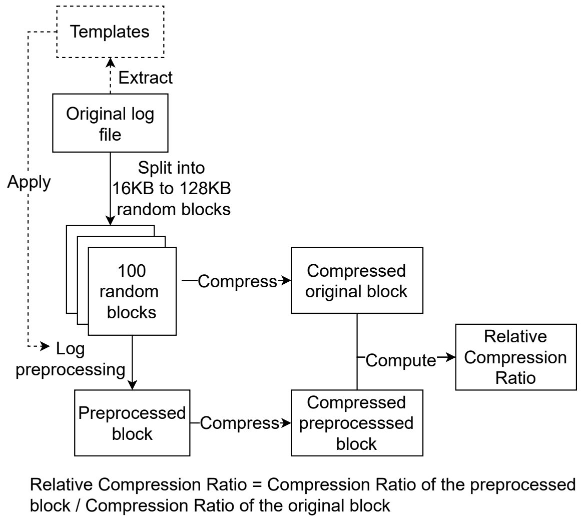

Templates
Extract
Original log file
Apply
Split into 16KB to 128KB random blocks
100 random blocks
Compress
Compressed original block
Log preprocessing
Compute
Relative Compression Ratio
Preprocessed block
Compress
Compressed preprocesssed block
Relative Compression Ratio = Compression Ratio of the preprocessed block / Compression Ratio of the original block

Fig. 3: The followed steps for evaluating the compression performance among *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* preprocessing and without preprocessing. The steps marked in dashed lines are the extra steps needed in *Logzip-E*.

and number of parameters) and their relative compression ratio. We use Spearman correlation because it does not have a presumption on the normality of the studied data [48]. We use four measures to describe the characteristics of each log block, which are: the unique number of log templates, the unique number of parameters (the parameters that are used to distinguish a log line from the rest of similar lines that are represented by the same template), the size of log templates, and the size of parameters.

Result: **Preprocessing the blocks of most log files using *Logzip* makes the compressed size larger than compressing the same blocks without any preprocessing.** As shown in Figure 2, the relative compression ratios of both *Logzip* and *Logzip-E* are always below the baseline (compression without any preprocessing) on all the 16KB log blocks. The compression ratios of *Logzip* and *Logzip-E* are 4% to 98% (with a median of 34%) and 4% to 99% (with a median of 31%) of the compression ratio without preprocessing on all 16KB log blocks. In addition, the compression ratios of *LogArchive* and *Cowic* are 72% to 78% and 12% to 15% of the compression ratio of directly using the general compressors without preprocessing. In the prior work, the compression ratio achieved by *Logzip* using *gzip* on five large-sized log files ranged from 160% to 1,510% of the compression ratio without preprocessing [27]. The compression ratio achieved by *LogArchive* is improved by 30% on an Apache web log compared to the compression ratio without preprocessing [32]. *Cowic* has a comparable compression ratio (up to 4%) compared to *gzip* [31]. However, all of the compression improvement in these approaches are achieved with large log files. Therefore, the aforementioned results indicate that although state-of-the-art log compression approaches improve the compression ratio on large-sized log files, they do not perform well on small-sized log blocks (the actual compression use cases of modern log management tools).

Despite leveraging the global repetitiveness, *Logzip-E* does not have a better compression performance for small log blocks over *Logzip*. *Logzip-E* was expected to outperform *Logzip* in both compression ratio and compression speed since the template extraction step is skipped and the template data is stored externally, as shown in Figure 3. However, we observe from Figure 2 that *Logzip* and *Logzip-E* have a similar compression ratio across the different log files. Furthermore, we do not observe statistically significant differences between the compression performance of *Logzip* and *Logzip-E*. The result indicates that template extraction in small logs consumes only a negligible amount of time. We examine the results and find that although the global template from *Logzip-E* may contain more accurate log templates (e.g., accurately identifying a parameter in logs) than the local template to a log block that is obtained using *Logzip*, such advantage does not result in a better compression ratio. For example, the template specific parameters will be further split and stored in different files, introducing additional overhead. Therefore, we consider the original *Logzip* as a more optimal approach than *Logzip-E* in compressing small log blocks. In particular, even with a similar compression ratio and speed, *Logzip-E* requires additional cost due to the extraction of log templates from large-scale log data. More importantly, due to the ever-changing nature of logs in large software systems [49], these pre-extracted log templates are very likely to become outdated and would require additional effort to maintain. On the other hand, such additional cost and effort do not demonstrate promising benefits in compression ratio and speed over *Logzip* in our results.

The unique number of templates of a log block has a moderate association with the relative compression ratio using *Logzip-E*. We observe a negative moderate correlation (-0.51, -0.43 and -0.56 correlation using *gzip*, *lz4* and *deflate* compressors) between the number of unique log templates on a log block and its relative compression ratio achieved by *Logzip-E*. This implies that blocks with a higher compression ratio may be associated with a low number of unique log templates. That indicates the existence of redundant data, which can be summarized in a few templates. For example, the HPC log blocks gain the highest compression ratio increment among all log blocks. By manually checking the HPC log blocks with the highest relative compression ratio, we observe that there exists no log content (only with the non-content parts, such as timestamps and log levels) inside those log blocks. For the other log files, the log blocks with the highest compression ratio contain fewer templates than those with the lowest compression ratio. For instance, the 32KB Windows log blocks with the highest relative compression ratio contain 6, 5 and 10 templates, while the blocks with the lowest relative compression ratio contain 23, 21 and 18 templates. Similarly, we observe that the majority of the log blocks are classified into a large number of buckets due to their insimilarity.

In *Logzip*, the extracted parameters will be stored in different files according to the associated log template, location of the parameter and the location of a token inside the parameter. Thus, for complex log blocks with diverse log templates, the parameter information is saved into hundreds of separate files, while the similarities within these files are not leveraged by general compressors. Hence, *Logzip* does not perform well on small log blocks. Similarly, for *LogArchive*, because of the low content repetitiveness in small log blocks, the majority of the logs are split into more than five buckets. Hence, *LogArchive* does not perform well when the general compressors are used to compress multiple buckets with small sizes. The need for additional storage for bucketing indexes also introduces overhead to the compressed data. Finally, for *Cowic*, the models learned from the small size of log blocks may not perform well to predict the rest of the logs, leading to a compression ratio that is worse than the general compressors. In short, none of the studied existing approaches perform well due to the characteristics of the small log blocks.

On the other hand, we observe that the size of the non-content part of a log block takes a median of 44.7% and as much as 85.6% of the total size of our studied log blocks. In prior log preprocessing studies [29, 31], log data is treated as separate columns. However, these prior studies are either applicable to log data with specific formats [29] or focus on the compression of line-level log data [31], without considering improving the compression of block-level log data. Inspired by prior work, we propose a column-based preprocessing approach that focuses on the non-content part of log blocks.

Summary

The state-of-the-art log compression approaches do not perform well on small log blocks (the actual compression scenarios in use by log management tools). Leveraging the global repetitiveness may further impair the compression performance of small log blocks. This suggests the need for a more efficient approach for preprocessing small blocks of log files.

5 *LogBlock*: Our Log Preprocessing Approach

From our preliminary study, we observe that *Logzip* does not perform well on compressing small log blocks. Hence, we propose a log preprocessing approach named *LogBlock* that specifically leverages the characteristics of small log blocks.

Unlike large-sized log files that have globally repetitive log content (log templates occur repeatedly at different locations throughout the whole file), the content part of small log blocks is less repetitive. Introducing additional pattern extraction or bucketing steps for less repetitive data would introduce additional overhead (e.g., computing and storage for the indexes) to the compressed result. Therefore, we choose to focus only on preprocessing the repetitiveness within the non-content part of logs.

From an initial investigation, we observe that within the non-content part of log lines, there exist four types of repetitiveness, which are identical tokens, similar numeric tokens, repetitive tokens and tokens with a common prefix string. Thus, our preprocessing approach intends to reduce the redundant information from these four types of repetitiveness.

- **Identical tokens (T1):** Since the log lines in small log blocks are generated in a short period of time (e.g., within the same minute), columns (e.g., Year, Day, Hour) are likely to contain the same information within small log blocks. For example, the *Year* column in Figure 6 has an identical token 2005.
- **Similar numeric tokens (T2):** Since logs in small log blocks are printed in a short period of time, there exist columns (e.g., timestamp) with non-identical but similar tokens in a small log block. For example, as shown in Figure 6, the tokens in the *TimeStamp* column have a maximum of a single-digit difference.
- **Repetitive tokens (T3):** We observe columns in log blocks that consist of repeating tokens. For instance, the *Log level* component in Figure 6 contains only three repeated unique tokens (INFO, WARN and ERROR).
- **Tokens with a common prefix string (T4):** Since logs in small log blocks are printed in a short period of time, we observe columns that contain tokens with the same prefix string. For example, the *Module* column in Figure 6 consists module names from the same parent module (*org.apache.hadoop.*).

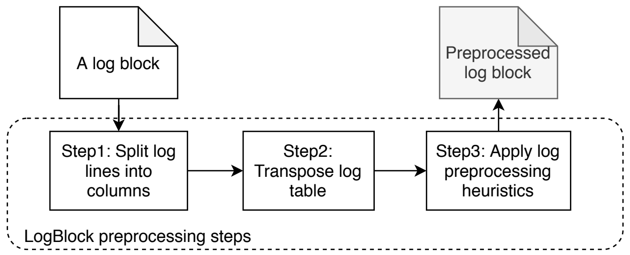
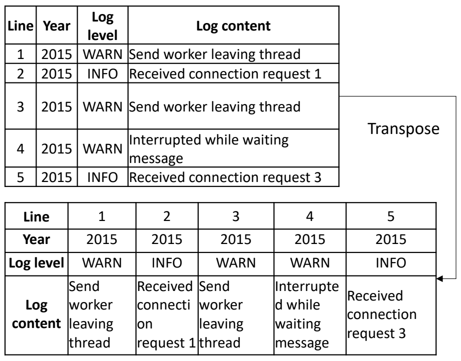

A log block
Preprocessed log block
Step1: Split log lines into columns
Step2: Transpose log table
Step3: Apply log preprocessing heuristics
LogBlock preprocessing steps

Fig. 4: The log preprocessing steps of *LogBlock*.

Step1: Split log lines into columns. In prior research [27, 31], the different headers (e.g., Month, Day, Hour) of log data are split into columns using a pre-defined log format. The log format can be either manually defined by developers or automatically extracted from log configuration files. For example, in Log4J2 [40], the format of a logging statement can be specified as *%d{yyyy-MM-dd}-%p-%c{1}-%m%n*. By parsing such a format, we can automatically identify the column as *Year*, *Month*, *Day*, *Log level*, *Logger* and *Log Content*. Similar tokens are grouped by columns to facilitate further processing. In this step, for each log block, we also split the log lines into columns. Figure 4 shows a real example of five columns (Year, Timestamp, Log level, Module, and Log content).

There may exist log lines that cannot be matched by a pre-defined log format. For instance, a log line starts with the content of a run-time exception, as shown in the example of Figure 6 (line 8 and line 9). We concatenate the adjacent unmatched lines together then append them to the content of their preceding log line (line 7). Thus, each log block can be regarded as a table where rows are formed by different log lines.

Step2: Transpose log table. Data encoding algorithms like *LZ77* are based on a fix-sized sliding window. When encoding a new token, such algorithms will search for this token using a sliding window then refer the new token to its previous occurrence. Transposing will gather similar tokens which can be easily referable by the sliding window encoder. Afterward, we transpose the table so that similar information from the same column would be put closer to each other, as shown in Figure 5. The transposing approach was used in the prior work to improve the compression of certain kinds of log files (i.e., Windows security logs, HTTP server log data) [35].



Line	Year	Log level	Log content
1	2015	WARN	Send worker leaving thread
2	2015	INFO	Received connection request 1
3	2015	WARN	Send worker leaving thread
4	2015	WARN	Interrupted while waiting message
5	2015	INFO	Received connection request 3

Line	1	2	3	4	5
Year	2015	2015	2015	2015	2015
Log level	WARN	INFO	WARN	WARN	INFO
Log content	Send worker leaving thread	Received connecti on request 1	Send worker leaving thread	Interrupte d while waiting message	Received connection request 3

Fig. 5: An example of transposing log table.

Step3: Apply preprocessing heuristics. We further improve the compression ratio by applying four additional preprocessing heuristics that we apply to the transposed log table, as shown in Figure 6. Each type of repetitiveness will be automatically detected and reduced through its corresponding heuristic. Each of the preprocessing is associated with an aforementioned type of repetitiveness in small log blocks. We provide an illustrative example in Figure 6 to show the result of applying these different heuristics during the preprocessing of log blocks. We summarize our four heuristics as follows:

- **Extract identical tokens (H1):** For the columns with identical tokens (T1), we extract the identical tokens from these columns and count the occurrences of these tokens. For example, as shown in Figure 6, the columns with identical tokens (e.g., 7 lines with 2005) are represented by the identical token (2005) and its occurrence count (7). Although T1 and T3 in theory are similar, H1 is implemented without encoding to save the overhead for dictionary and indexing.
- **Delta encoding for numbers (H2):** For those numeric columns with similar tokens (T2), we adopt delta encoding by referring to the delta between the current token and its prior token. The first token in each column is retained. Thus, a token with long text is replaced with a shorter representation.
- **Build dictionary for repetitive tokens (H3):** For the columns that contain few repetitive tokens (T3), we build a dictionary for each identical token and replace the tokens with a shorter representation (corresponding index from the dictionary), as shown in Figure 6.
- **Extract a common prefix string (H4):** For the columns with a common prefix string (T4), we extract the prefix string and only store the remaining part of each token. In particular, we extract the common prefix string from the list by comparing characters from the beginning until different characters are observed. As shown in Figure 6, the common prefix string *org.apache.hadoop* is extracted from the column. This heuristic removes redundant information from each token to make it shorter.

All of the processed columns are output to a single file. Finally, we compress each preprocessed log block using one of the studied compressors (i.e., *gzip*, *lz4* and *deflate*).

Modern log management tools usually process log data by splitting log data or log streams into small blocks and indexing them for future queries. When a query is performed, the blocks that are identified by their indexes will be decompressed. *LogBlock* is based on lossless compression and would not impact the way that each small block of log data is indexed and searched. The compression process remains a blackbox step for log querying. Therefore, *LogBlock* will not influence the existing log query process.

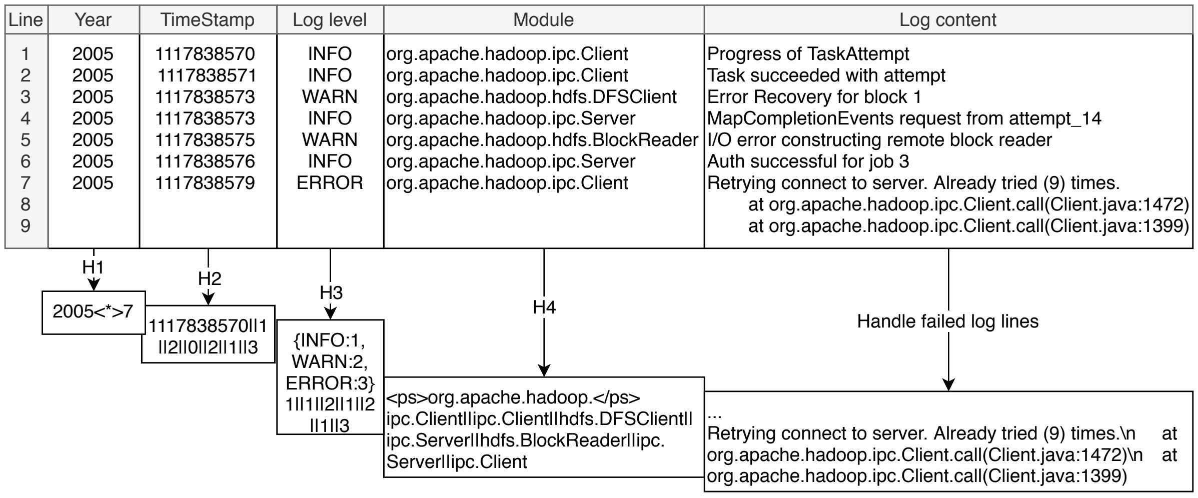
6 Evaluation

In this section, we first evaluate the performance of *LogBlock* on compressing small blocks of log files compared to *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* (the state-of-the-art approaches for compressing log files). Then we investigate the impact of each preprocessing heuristic of *LogBlock* on the compression of small log blocks.

We use the same experimental setup that we used to evaluate *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* (Preliminary Study), which is summarized in Figure 3. We compare the compression ratio between the blocks without any preprocessing and the same blocks preprocessed by *LogBlock*. Then for each of the preprocessing heuristic, we exclude one heuristic from *LogBlock* at a time and compare the compression performance between preprocessing with a heuristic excluded and preprocessing with all heuristics, as shown in Figure 7. Then we use the Wilcoxon rank-sum test [47] ($\alpha = 0.05$) to examine whether the compression performance is statistically different before and after excluding each heuristic. The Wilcoxon rank-sum test is used to determine whether two distributions of compression performance metrics are statistically different from each other. We choose the Wilcoxon rank-sum test because it does not have any assumption on the distribution of the data. Note that we use the same blocks as the preliminary study with sizes of 16KB, 32KB, 64KB, 128KB. The compression performance is evaluated using *deflate*, *gzip* and *lz4* compressors.

6.1 Evaluate the compression performance of *LogBlock*.

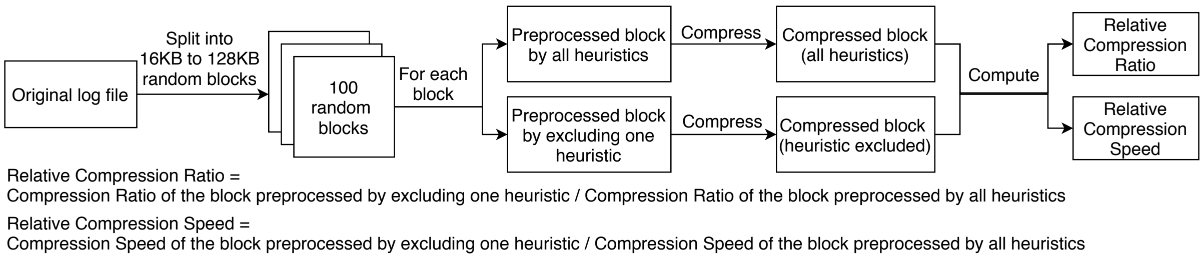
Our approach improves the compression ratio by a median of 5%, 9%, 15% and 21% for 16KB, 32KB, 64KB, and 128KB blocks in comparison to compression without any preprocessing. As shown in Figure 8, the compression ratios are improved with the *LogBlock* preprocessing. The ELK Stack log management tool splits logs into 16KB blocks then compresses them using *lz4*. In particular, *LogBlock* improves the relative compression ratio by a median of 6% when logs are split into 16KB blocks then compressed using *lz4* (default setting of ELK Stack). On the other hand, *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* have a median of 67% to 38%, 71% to 44%, 28% to 10% and 88% to 85% lower relative compression ratio than their compression without preprocessing on the studied block sizes. To sum up, *LogBlock* has 1.7 to 8.4 times, 1.9 to 10.0 times, 1.3 to 1.9 times and 6.2 to 11.4 times higher compression ratio than *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* on small log blocks.



Line	Year	TimeStamp	Log level	Module	Log content
1	2005	1117838570	INFO	org.apache.hadoop.ipc.Client	Progress of TaskAttempt
2	2005	1117838571	INFO	org.apache.hadoop.ipc.Client	Task succeeded with attempt
3	2005	1117838573	WARN	org.apache.hadoop.hdfs.DFSClient	Error Recovery for block 1
4	2005	1117838573	INFO	org.apache.hadoop.ipc.Server	MapCompletionEvents request from attempt_14
5	2005	1117838575	WARN	org.apache.hadoop.hdfs.BlockReader	I/O error constructing remote block reader
6	2005	1117838576	INFO	org.apache.hadoop.ipc.Server	Auth successful for job 3
7	2005	1117838579	ERROR	org.apache.hadoop.ipc.Client	Retrying connect to server. Already tried (9) times.
8					at org.apache.hadoop.ipc.Client.call(Client.java:1472)
9					at org.apache.hadoop.ipc.Client.call(Client.java:1399)

```
2005<*>7

1117838570ll1
ll2ll0ll2ll1ll3

{INFO:1,
WARN:2,
ERROR:3}
1ll1ll2ll1ll2
ll1ll3

<ps>org.apache.hadoop.</ps>
ipc.Clientllipc.Clientllhdfs.DFSClientll
ipc.Serverllhdfs.BlockReaderllipc.
Serverllipc.Client

...
Retrying connect to server. Already tried (9) times.\n    at
org.apache.hadoop.ipc.Client.call(Client.java:1472)\n    at
org.apache.hadoop.ipc.Client.call(Client.java:1399)
```

Fig. 6: An example of how *LogBlock* preprocesses log data.


Original log file
Split into 16KB to 128KB random blocks
100 random blocks
For each block
Preprocessed block by all heuristics
Preprocessed block by excluding one heuristic
Compress
Compress
Compressed block (all heuristics)
Compressed block (heuristic excluded)
Compute
Relative Compression Ratio
Relative Compression Speed

Relative Compression Ratio =
Compression Ratio of the block preprocessed by excluding one heuristic / Compression Ratio of the block preprocessed by all heuristics

Relative Compression Speed =
Compression Speed of the block preprocessed by excluding one heuristic / Compression Speed of the block preprocessed by all heuristics

Fig. 7: The evaluation steps between *LogBlock* with all heuristics and *LogBlock* with one heuristic excluded. The relative compression ratio (relative compression speed) is the compression ratio (compression speed) from compression with all heuristics over compression with one heuristic excluded, respectively. Note that we measure the elapsed time for both preprocessing and compressing when evaluating the compression speed.

***LogBlock* has a drastically higher compression speed than the state-of-the-art log compression approaches.** When comparing the compression speed between *LogBlock* and the state-of-the-art log compression approaches, *LogBlock* is still 30.8 to 49.7 times faster than *Logzip*, 42.6 to 53.8 times faster than *Logzip-E*, 4.5 to 6.0 times faster than *LogArchive* and 5.0 to 19.3 times faster than *Cowic* in preprocessing and compressing small-sized log blocks.

There exist a few cases where *Logzip* has a higher compression ratio than LogBlock in certain blocks. We observe that the majority of these blocks are from the Proxifier log. After manually checking the top 10 blocks with the largest differential in compression ratio, we observe that these blocks contain fewer templates than other blocks of the same log file for the same block size. For example, 128KB Proxifier blocks contain 7.9 templates on average, while for the blocks where *Logzip* outperforms, the blocks only have 3 or 4 templates.

6.2 The impact of each preprocessing heuristic of *LogBlock* on the compression performance.

***Delta encoding for numbers (H2)* has the largest improvement in relative compression ratio among all heuristics, as shown in Table 3. *H2* improves the compression ratio by a median of 4.7%, 5.1% and 4.9% (and as much as 40.1%, 48.5% and 41.6%) with *deflate*, *gzip* and *lz4*.** *H2* effectively improves the relative compression ratio for the HPC log, the OpenSSH log and the BGL log, as shown in Table 3. From a manual investigation, we observe that these log files usually contain one or few columns with long numeric text, such as timestamp and process id. Such columns are likely to be similar in a small log block. Hence, this heuristic reduces the amount of information by replacing long numeric text with a digit. Besides, we observe that the relative compression speed is decreased by a median of 7.7%, 6.4% and 8.0% with *deflate*, *gzip* and *lz4*.

TABLE 3: The impact on the relative compression ratio loss and compression speed increment from excluding each preprocessing heuristic.

Exclude Heuristic	Compressor	Ratio Loss (%)	Speed Increase (%)
H1	deflate	1.3	-12.0
	gzip	1.5	-2.3
	lz4	0.9	-5.0
H2	deflate	4.7	7.7
	gzip	5.1	6.4
	lz4	4.9	8.0
H3	deflate	1.7	4.6
	gzip	1.7	5.5
	lz4	2.5	6.3
H4	deflate	0.0	6.2
	gzip	0.0	5.3
	lz4	1.3	6.7

***Extract unique tokens (H1)* improves the relative compression ratio by a median of 1.3%, 1.5% and 1.0% (and**

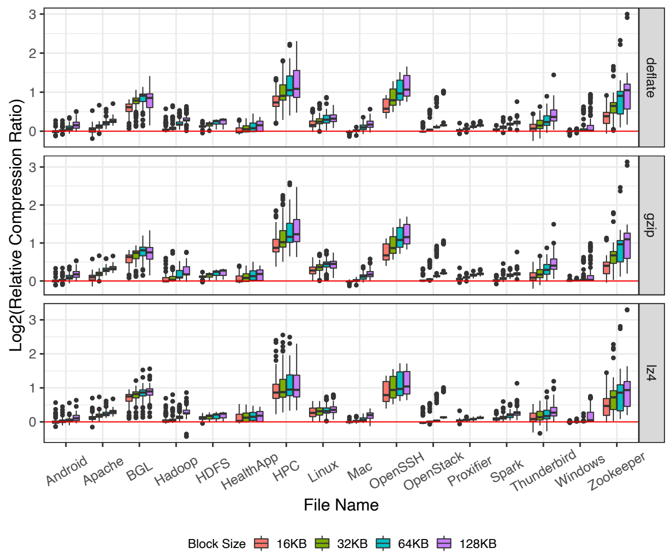


Fig. 8: The relative compression ratio (in log-scale) between compression without and with preprocessing using *LogBlock*. Each box shows the distribution of relative compression ratio of 100 randomly selected blocks from the same log file using a specific block size (16KB, 32KB, 64KB and 128KB) and compressor (*gzip*, *lz4* and *deflate*). The horizontal red line at zero ($\log_2 1$) is the baseline that indicates the relative compression ratio without preprocessing. The compression ratio values can be found in Table 5.

as much as 6.0%, 6.4% and 2.9%) with *deflate*, *gzip* and *lz4*. In particular, as shown in Table 3, *H1* has a higher relative compression ratio increment (4.8%, 2.5% and 2.1% by median) at BGL, Thunderbird and Spark log files. For example, *H1* improves the relative compression ratio for the BGL log by 0.2% to 15.1%. On the other hand, we observe that excluding *H1* may sometimes even lead to a better relative compression ratio (e.g., compressing 128KB HPC log blocks using *lz4*.) By manually checking those log blocks we observe that the columns can still be processed by *H2* if *H1* is excluded. Thus, the increased compression ratio is caused by processing such columns with a different heuristic.

Extract unique tokens (H1) improves the relative compression speed, as shown in Table 3. Adopting the *H1* preprocessing heuristic improves the compression speed by a median of 12.0%, 2.3% and 5.0% (and as much as 22.9%, 14.3% and 14.3%) with *deflate*, *gzip* and *lz4*. Since the information in the columns of small log blocks is likely to have no variance, preprocessing is likely to be faster than compression algorithms on columns with unique tokens.

***Build dictionary for repetitive tokens (H3)* and *Extract a common prefix string (H4)* improves the compression ratio as much as 22.2% and 9.7% for certain log files, respectively.** For specific columns (e.g., log level, module, as shown in Figure 6) in small log blocks, *H3* and *H4* reduce the information by either replacing tokens with shorter representations or reduce information from each token. We observe that *H3* and *H4* effectively improve the relative compression ratio of the Zookeeper and BGL logs respectively. In particular, *H3* and *H4* improve the relative compression ratio by a median of 14.5% and 3.4% at the cost of a median of 2.9% and 5.3% slowdown in relative compression speed on Zookeeper log and BGL log with *lz4*, respectively. From a manual investigation, we observe that the Zookeeper log contains three columns with repetitive tokens and the BGL log contains three columns with common prefix strings. Thus, the *H3* and *H4* preprocessing heuristics achieve a better relative compression ratio at these files. Besides, the relative compression speed is improved at 75% and 57% of the log files using *H3* and *H4*. This implies that these preprocessing heuristics might not be applicable to all types of log files.

Summary

LogBlock improves the relative compression ratio by a median of 5% to 21% comparing to general compressors. When compared with state-of-the-art log compression approaches, *LogBlock* outperforms *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* by a median of 1.7 to 8.4 times, 1.9 to 10.0 times, 1.3 to 1.9 times and 6.2 to 11.4 times in compression ratio, respectively, and a median of 30.8 to 49.7 times, 42.6 to 53.8 times, 4.5 to 6.0 times and 2.5 to 4.0 times in compression speed at different block sizes. *Delta encoding for numbers (H2)* and *Extract unique tokens (H1)* produce the largest improvement in compression ratio and compression speed respectively. Practitioners should build their log preprocessing pipeline based on *H1* and *H2*, then customize the other preprocessing heuristics according to the characteristics of their own log data.

7 Discussion

In this section, we discuss the compression performance of *LogBlock* and *Logzip* on different block sizes. Our evaluation results show that *LogBlock* has a better compression performance in small log blocks, while prior research [27] demonstrates the superior compression performance of *Logzip* with large logs. Therefore, we would like to investigate to what extent does *LogBlock* outperform *Logzip* in compression ratio when increasing the sizes of log blocks.

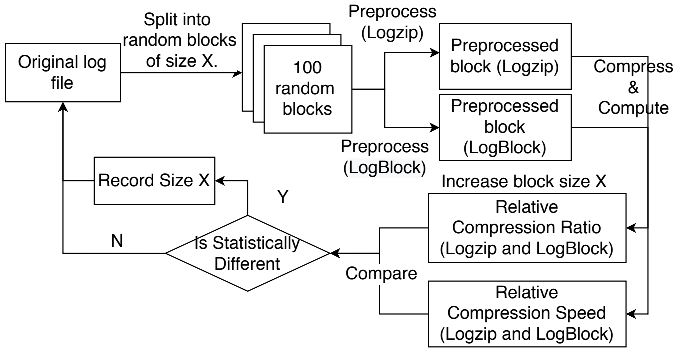

Original log file
Split into random blocks of size X.
100 random blocks
Preprocess (Logzip)
Preprocessed block (Logzip)
Preprocess (LogBlock)
Preprocessed block (LogBlock)
Compress & Compute
Increase block size X
Relative Compression Ratio (Logzip and LogBlock)
Relative Compression Speed (Logzip and LogBlock)
Compare
Is Statistically Different
Y
N
Record Size X

Fig. 9: The evaluation steps of the compression performance between *LogBlock* and *Logzip*. We evaluate the block size X increasingly from 16KB to 32MB.

We iteratively increase the log block sizes and compare the performance of our approach to *Logzip*, as summarized in Figure 9. Since we mainly focus on improving the compression ratio on small-sized log files, we only evaluate block sizes from 16KB up to 32MB. We stop increasing the block size of a log file when *Logzip* statistically (Wilcoxon rank-sum test: $\alpha = 0.05$) outperforms *LogBlock* in terms of compression ratio. We also evaluate the compression performance using the *deflate*, *gzip*, and *lz4* compressors.

TABLE 4: The block sizes that *Logzip* begins to outperform *LogBlock* in compression ratio.

File	Compressor	Chunk size
Android	gzip	32M
BGL	gzip	16M
	lz4	32M
HDFS	deflate	32M
	gzip	32M
	lz4	32M
HealthApp	gzip	2M
	lz4	2M
	deflate	4M
Mac	gzip	16M
	lz4	16M
Proxifier	gzip	512K
	lz4	512K
Spark	deflate	16M
	gzip	1M
Windows	gzip	1M
	lz4	256K
	deflate	4M

* For the rest 30 combinations of log file and general compressors that are not listed in the table, *LogBlock* always has higher compression ratios than *Logzip*.

The block sizes at which *Logzip* outperforms *LogBlock* in terms of compression ratio are not the sizes that are adopted by popular log management tools. We observe from Table 4 that *Logzip* begins to outperform *LogBlock* at a median of 10MB, 9MB and 9MB block size with *deflate*, *gzip* and *lz4* compressors, respectively. However, none of these block sizes is used by popular log management tools (i.e., 16KB used by ELK Stack, 128KB used by Splunk). In addition, we observe that the *deflate* compressor requires an equal or larger block size than the *gzip* compressor when *Logzip* begins to outperform *LogBlock*.

For specific log files, *Logzip* slightly increases the compression ratio with a drastic sacrifice of the compression speed. For example, we observe that when preprocessing 512KB Proxifier log blocks using *Logzip* with *gzip*, the compression speed slows down by 21.5% with only 2.9% improvement in compression ratio when compared to *LogBlock*. Similarly, when preprocessing 256KB blocks of Windows log using *lz4*, *Logzip* sacrifices 14.6% compression speed to achieve merely 0.8% compression ratio improvement. ***Logzip* has a median of 14.7% higher compression ratio and a median of 14.2% lower compression speed than *LogBlock*.** When *Logzip* starts to outperform *LogBlock* in compression ratio, switching preprocessing approach from *LogBlock* to *Logzip* could gain additional compression ratio but also lead to a slowdown in compression speed. Thus, we suggest that practitioners choose their log preprocessing approach according to their log files and their desired block sizes.

Summary

When evaluating the compression ratio of preprocessed blocks, *LogBlock* outperforms *Logzip* for all block sizes that are used by popular log management tools. *Logzip* outperforms *LogBlock* when preprocessing blocks with a median size of 9MB.

8 Threats to Validity

External validity. In this paper, we evaluate the compression performance of 16 log files from different software systems. In particular, we use the block sizes (16KB, 32KB, 64KB, and 128KB) with specific compressors (*gzip*, *lz4* and *deflate*) that are commonly used in modern log management tools, while our evaluation results are quite consistent across the studied system, they may not generalize to other settings. However, our proposed approach can be replicated by others for their own settings.

Internal validity. An internal threat to validity considers the evaluation approach, which might be impacted by the characteristics of the evaluated log blocks. In fact, a block might benefit more from *LogBlock* depending on its content. To mitigate this risk, we consider in our evaluation a large number of blocks (i.e., 100 blocks for each case study) that are randomly sampled.

9 Conclusion

Modern log management tools split large-sized log data into small blocks before compressing each block separately. While a large body of research effort focuses on compressing large log files, no prior research focuses on the compression performance of small log blocks. In this paper, we first evaluate the impact of four state-of-the-art log preprocessing approaches (i.e., *Logzip* and its variation *Logzip-E*, *LogArchive* and *Cowic*) on the compression of small log blocks. We observe that these approaches do not perform well on small log blocks with a median of 52.1%, 16.2% and 87.5% lower compression ratio than compression without preprocessing. Hence, we propose a log preprocessing approach (*LogBlock*) that considers the characteristics of small log blocks. Evaluating *LogBlock* on 16 log files, we observe that *LogBlock* improves the compression ratio by a median of 5%, 9%, 15% and 21% on 16KB, 32KB, 64KB and 128KB log blocks, which outperforms *Logzip*, *Logzip-E*, *LogArchive* and *Cowic* by 1.7 to 8.4 times, 1.9 to 10.0 times, 1.3 to 1.9 times and 6.2 to 11.4 times in compression ratio, 30.8 to 49.7 times, 42.6 to 53.8 times, 4.5 to 6.0 times and 2.5 to 4.0 times in compression speed. Our approach can help improve the storage efficiency of log management tools.

References

[1] H. Li, W. Shang, Y. Zou, and A. E. Hassan, "Towards just-in-time suggestions for log changes," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1831–1865, 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9467-z

[2] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs

of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, 2020.

[3] Y. Zeng, J. Chen, W. Shang, and T.-H. P. Chen, "Studying the characteristics of logging practices in mobile apps: a case study on f-droid," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3394–3434, Dec 2019. [Online]. Available: https://doi.org/10.1007/s10664-019-09687-9

[4] K. Yao, G. B. de Pádua, W. Shang, C. Sporea, A. Toma, and S. Sajedi, "Log4perf: suggesting and updating logging locations for web-based systems' performance monitoring," *Empirical Software Engineering*, vol. 25, no. 1, pp. 488–531, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09748-z

[5] W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding log lines using development knowledge," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 21–30. [Online]. Available: https://doi.org/10.1109/ICSME.2014.24

[6] Q. Fu, J. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie, "Contextual analysis of program logs for understanding system behaviors," in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, 2013, pp. 397–400. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624054

[7] Q. Fu, J. Lou, Y. Wang, and J. Li, "Execution anomaly detection in distributed systems through unstructured log analysis," in *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, 2009, pp. 149–158. [Online]. Available: https://doi.org/10.1109/ICDM.2009.60

[8] S. He, J. Zhu, P. He, and M. R. Lyu, "Experience report: System log analysis for anomaly detection," in *27th IEEE International Symposium on Software Reliability Engineering, ISSRE 2016, Ottawa, ON, Canada, October 23-27, 2016*, 2016, pp. 207–218. [Online]. Available: https://doi.org/10.1109/ISSRE.2016.21

[9] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 102–112.

[10] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *Journal of Software Maintenance and Evolution: Research and Practice*, pp. 249–267, 2008.

[11] V.-A. Zamfir, M. Carabas, C. Carabas, and N. Tapus, "Systems monitoring and big data analysis using the elasticsearch system," in *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*. IEEE, 2019, pp. 188–193.

[12] O. Kononenko, O. Baysal, R. Holmes, and M. W. Godfrey, "Mining modern repositories with elasticsearch," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 328–331.

[13] A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield, and D. Dietz, "An industrial case study of customizing operational profiles using log compression," in *Proceedings of the 30th international conference on Software engineering*, 2008, pp. 713–723.

[14] What is the ELK stack? (Accessed on 07/04/2019). [Online]. Available: https://www.elastic.co/elk-stack

[15] Siem, aiops, application management, log management, machine learning, and compliance. (Accessed on 07/04/2020). [Online]. Available: https://www.splunk.com/

[16] Store compression in lucene and elasticsearch. (Accessed on 07/04/2020). [Online]. Available: https://www.elastic.co/blog/store-compression-in-lucene-and-elasticsearch

[17] J. Champagne. Behind the magnifying glass: How search works. (Accessed on 07/04/2020). [Online]. Available: https://static.rainfocus.com/splunk/splunkconf18/sess/1523558790516001KFjM/finalPDF/Behind-The-Magnifying-Glass-1734_1538786592130001CBKR.pdf

[18] Splunk enterprise and cloud | pricing | splunk. (Accessed on 12/21/2020). [Online]. Available: https://www.splunk.com/en_us/software/pricing/enterprise-and-cloud.html

[19] Lucene87storedfieldsformat. (Accessed on 12/21/2020). [Online]. Available: https://lucene.apache.org/core/8_7_0/core/org/apache/lucene/codecs/lucene87/Lucene87StoredFieldsFormat.html

[20] Log management security analytics, continuous intelligence | sumo logic. (Accessed on 07/04/2020). [Online]. Available: https://www.sumologic.com/

[21] syslog-ng - log management solutions. (Accessed on 07/04/2020). [Online]. Available: https://www.syslog-ng.com/

[22] Nginx | high performance load balancer, web server, reverse proxy. (Accessed on 07/04/2020). [Online]. Available: https://www.nginx.com/

[23] The rocket-fast syslog server - rsyslog. (Accessed on 07/04/2020). [Online]. Available: https://www.rsyslog.com/

[24] Cloud monitoring as a service | datadog. (Accessed on 07/04/2020). [Online]. Available: https://www.datadoghq.com/

[25] Sentry: Application monitoring and error tracking software. (Accessed on 07/04/2020). [Online]. Available: https://sentry.io/welcome/

[26] K. Yao, H. Li, W. Shang, and A. E. Hassan, "A study of the performance of general compressors on log files." Springer, 2020.

[27] J. Liu, J. Zhu, S. He, P. He, Z. Zheng, and M. R. Lyu, "Logzip: Extracting hidden structures via iterative clustering for log compression," *arXiv preprint arXiv:1910.00409*, 2019.

[28] F. J. Otten, "Using semantic knowledge to improve compression on log files," Ph.D. dissertation, Rhodes University, 2008.

[29] R. Balakrishnan and R. K. Sahoo, "Lossless compression for large scale cluster logs," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, p. 435.

[30] B. Feng, C. Wu, and J. Li, "MLC: an efficient multi-level log compression method for cloud backup systems," in *2016 IEEE Trustcom/BigDataSE/ISPA, Tianjin, China, August 23-26, 2016*, 2016, pp. 1358–1365. [Online]. Available:

https://doi.org/10.1109/TrustCom.2016.0215

[31] H. Lin, J. Zhou, B. Yao, M. Guo, and J. Li, "Cowic: A column-wise independent compression for log stream analysis," in *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2015, Shenzhen, China, May 4-7, 2015*, 2015, pp. 21–30. [Online]. Available: https://doi.org/10.1109/CCGrid.2015.45

[32] R. Christensen and F. Li, "Adaptive log compression for massive log data." in *SIGMOD Conference*. ACM, 2013, pp. 1283–1284.

[33] Lucene50storedfieldsformat. (Accessed on 07/04/2020). [Online]. Available: http://lucene.apache.org/core/7_7_0/core/org/apache/lucene/codecs/lucene50/Lucene50StoredFieldsFormat.html

[34] P. Skibiński and J. Swacha, "Fast and efficient log file compression," in *CEUR Workshop Proceedings of the 11th East-European Conference on Advances in Databases and Information Systems*, ser. ADBIS'07. ACM, 2007, pp. 330–342.

[35] P. Mell and R. E. Harang, "Lightweight packing of log files for improved compression in mobile tactical networks," in *Military Communications Conference (MILCOM), 2014 IEEE*. IEEE, 2014, pp. 192–197.

[36] K. Hätönen, J. F. Boulicaut, M. Klemettinen, M. Miettinen, and C. Masson, "Comprehensive log compression with frequent patterns," in *International Conference on Data Warehousing and Knowledge Discovery*. Springer, 2003, pp. 360–370.

[37] R. Surti and S. Saxena, "Littlelog: A log compression and querying service," 2018. [Online]. Available: https://rahulsurti97.github.io/littlelog.pdf

[38] G. Aceto, A. Botta, A. Pescapé, and C. Westphal, "Efficient storage and processing of high-volume network monitoring data," *IEEE Transactions on Network and Service Management*, vol. 10, no. 2, pp. 162–175, 2013.

[39] R. Agarwal, A. Khandelwal, and I. Stoica, "Succinct: Enabling queries on compressed data," in *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, 2015, pp. 337–350.

[40] Apache log4j 2. (Accessed on 07/04/2020). [Online]. Available: https://logging.apache.org/log4j/2.x/

[41] Logback project. (Accessed on 07/04/2020). [Online]. Available: https://logback.qos.ch/

[42] H. Li, W. Shang, B. Adams, M. Sayagh, and A. E. Hassan, "A qualitative study of the benefits and costs of logging from developers' perspectives," *IEEE Transactions on Software Engineering*, 2020.

[43] S. Yang, S. J. Park, and J. K. Ousterhout, "Nanolog: A nanosecond scale logging system," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, 2018, pp. 335–350. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/yang-stephen

[44] M. Marron, "Log++ logging for a cloud-native world," *ACM SIGPLAN Notices*, vol. 53, no. 8, pp. 25–36, 2018.

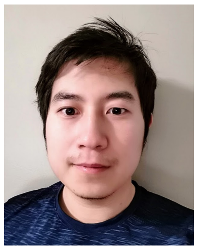
[45] Loghub: A large collection of system log datasets for ai-powered log analytics. (Accessed on 07/04/2020). [Online]. Available: https://github.com/logpai/loghub

[46] J. Zhu, S. He, J. Liu, P. He, Q. Xie, Z. Zheng, and M. R. Lyu, "Tools and benchmarks for automated log parsing," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP'19. IEEE Press, 2019, pp. 121–130.

[47] D. S. Moore, B. A. Craig, and G. P. McCabe, *Introduction to the Practice of Statistics*. WH Freeman, 2012.

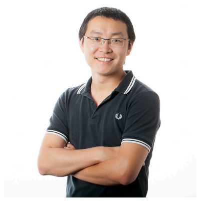
[48] T. D. V. Swinscow, M. J. Campbell *et al.*, *Statistics at square one*. Bmj London, 2002.

[49] S. Kabinna, C. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: a case study for the apache software foundation projects," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, M. Kim, R. Robbes, and C. Bird, Eds. ACM, 2016, pp. 154–164. [Online]. Available: https://doi.org/10.1145/2901739.2901769



Kundi Yao is a PhD student in the Software Analysis and Intelligence Lab (SAIL) at Queen's University, Canada. He obtained his M.sc. from Concordia University (Canada), and B.Eng. from Wuhan University of Technology (China). His research interests lie within software log compression, software log analytics, and mining software repositories. Contact him at kundi@cs.queensu.ca.

Mohammed Sayagh is an assistant professor at ETS (Quebec University). Before that, he was a postdoctoral fellow at the Software Analysis and Intelligence Lab (SAIL) at Queen's University. He obtained his PhD from the Lab on Maintenance, Construction, and Intelligence of Software (MCIS) at Ecole Polytechnique Montreal (Canada). His research interests include empirical software engineering, multi-component software systems, and software configuration engineering. More details about his work are available at https://msayagh.github.io.



Weiyi Shang is a Concordia University Research Chair at the Department of Computer Science. His research interests include AIOps, big data software engineering, software log analytics and software performance engineering. He is a recipient of various premium awards, including the SIGSOFT Distinguished paper award at ICSE 2013, best paper award at WCRE 2011 and the Distinguished reviewer award for the Empirical Software Engineering journal. His research has been adopted by industrial collaborators (e.g., BlackBerry and Ericsson) to improve the quality and performance of their software systems that are used by millions of users worldwide. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/~shang.

Ahmed E. Hassan is an IEEE Fellow, an ACM SIGSOFT Influential Educator, an NSERC Steacie Fellow, the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves/d on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. Contact ahmed@cs.queensu.ca. More information at: http://sail.cs.queensu.ca/.