LogPress: Optimized Compression and Retrieval of Unstructured Logs.

Neel Dudheliya
North Carolina State University
Raleigh, USA
ndudhel@ncsu.edu

Tanay Gandhi
North Carolina State University
Raleigh, USA
tgandhi@ncsu.edu

Pranav Jibhakate
North Carolina State University
Raleigh, USA
pjibhak@ncsu.edu

Abstract

In modern IT infrastructure, unstructured logs are expanding at an unprecedented level, and high compression combined with speedy searching becomes a great challenge. Conventional log management systems inefficiently store logs, which results in redundancy, high storage overhead, and slow retrieval. In this paper, we design a scalable log management system that effectively stores and searches unstructured console logs through pattern-aware structuring, optimized query processing, and adaptive compression mechanisms. Our method additionally incorporates clustering, chunking, and distributed compression: logs are initially clustered by algorithms such as K-means, then split into reasonable chunks for better compression, and subsequently compressed in a distributed fashion on worker nodes. By taking advantage of these methods, our system greatly lowers the storage expense while supporting quick querying without external indexes, making it extremely appropriate for big data log analytics and monitoring.

1 Introduction

Modern applications produce enormous amounts of unstructured log data, and this necessitates the requirement for effective storage and retrieval mechanisms. Various components such as applications, operating systems, cloud platforms, and network devices, constantly produce logs that contain useful information on system functionality, performance indicators, security incidents, and error tracking. The logs constitute the foundation for real-time diagnosis, anomaly detection, and auditing procedures, which are central in ensuring the stability and security of information technology infrastructures.

Unlike semi-structured or structured logs, unstructured logs do not have a pre-defined schema, making their storage, indexing, and analysis more challenging. These logs are typically a mix of a set of free text, timestamps, stack traces, error messages, and key-value pairs, blended in a variety of formats. This renders them highly redundant, making storage expensive, and challenging in extracting valuable information. The problem of storage and retrieval becomes much more significant as log volumes keep increasing. Large-scale organizations, such as Google, which claims to process more than 25 exabytes of log data annually [4], give an idea of the huge scale at which log management systems have to function. Log handling at such a massive scale demands advanced compression techniques to reduce the cost of storage and high-performance search mechanisms to enable fast log retrieval and analysis.

Where existing compression methods reduce storage needs, most of these must decompress completely to enable querying, and searches are slow and use too many resources. Again, regex-pattern and full-text searching over unstructured logs continue to be computationally expensive and not scalable, so there is still a tradeoff between storage efficiency and retrieval speed.

This research addresses these challenges by proposing a multiple novel systems like pattern-aware compression, clustering, chunking, distributed processing, and optimized search techniques. Our method records repetitive patterns in logs, expresses them in dictionary format, stores timestamps efficiently, and accommodates effective search queries without an external index. We further introduce a clustering option to group similar logs by means of K-means clustering, a chunking technique to divide logs into pieces to allow more effective compression, and a distributed system to employ a variety of worker nodes to parallel compress and search. These innovations helps us understand how our system to achieve high compression ratios while maintaining fast, scalable retrieval capabilities, making it highly effective for large-scale, unstructured log management.

2 Design and Algorithm

In this section, we will outline the design choices and algorithmic strategies implemented to address the challenges associated with efficient log compression and retrieval. Specifically, we elaborate on our methodology involving log clustering, template-based compression, and highlights their significance and operational intricacies. Additionally, we discuss how these methodologies can effectively integrate within a distributed architecture to ensure scalability and performance optimization across large-scale log management systems.

2.1 Chunking-based Approach

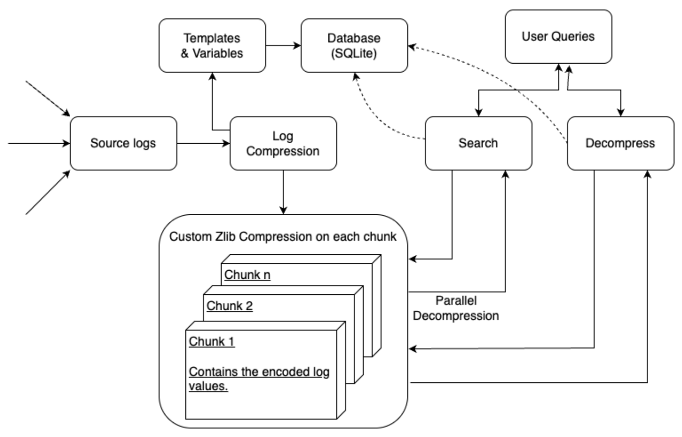


Figure 1: Chunking-based compression architecture

In this approach, we designed a log compression and retrieval system that employs template-based compression and multithreaded querying methodologies to efficiently store and access large-scale log data. This system is particularly designed to address the challenges associated with the storage and analysis of massive log files generated by modern computing systems.

The compression process begins with the parsing of log files to identify and extract templates. Each log entry is analyzed to detect variable components, which are then replaced with a standardized placeholder <VAR>, resulting in a template that represents a class of similar log messages. The unique templates and their corresponding variable values are stored separately, allowing for a significant reduction in redundancy and overall data size. Every log entry is then divided into a fix-sized chunks for individual zlib compression.

To compress more efficiently, we use the zlib library with a custom dictionary generated from the extracted variables and template values. This takes advantage of repeated patterns in log data to achieve better compression than standard methods. Compressed data blocks are saved with metadata linking them back to their original log entries for easy reconstruction when decompressed.

Our system uses a multithreaded searching process to scan and examine compressed logs. Once the system gets a search request, it starts some threads to explore different areas of the compressed data in parallel. This parallel decompression greatly minimizes latency in searching, making real-time query against large-scale data possible. In addition, the system can highlight keywords in the results to facilitate convenient access to useful information.

Our solution uses template-based compression and multithreaded searching to achieve an efficient and scalable solution for storing and retrieving log data. Our solution minimizes storage needs and improves the accuracy and speed of log analysis to meet the key needs of system administrators and developers dealing with enormous volumes of logging data.

2.2 Clustering-based Approach

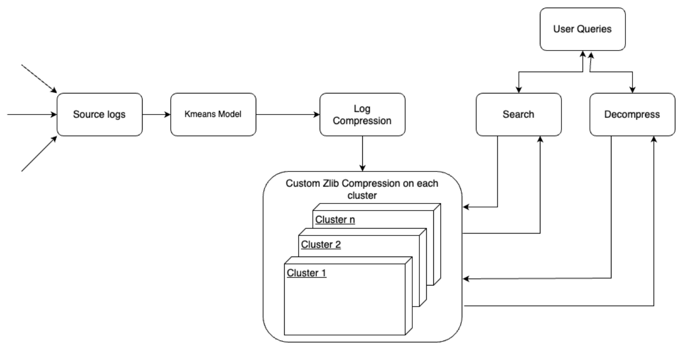


Figure 2: Clustering-based compression architecture

In our efficient log compression and retrieval system, we take an integrated, multi-step approach of clustering, template extraction, dictionary-based compression by custom dictionary, and multithreaded search mechanisms to manage large-scale log data sets efficiently.

We initially group all of the log data according to the K-means algorithm, a machine learning algorithm with a good track record in grouping data points into clusters based on their similarity. Unlike typical practices that simply split logs into arbitrary fixed-size chunks, our clustering method takes advantage of both the structure and text of log messages to form clusters of logs with high similarity within each cluster. This careful grouping significantly improves the performance of the following stages of compression by enabling algorithms to make full use of repeated patterns within the groups.

Subsequently, we perform template extraction on each cluster individually. Log entries frequently adhere to defined structural patterns characterized by fixed and variable components. The template extraction process begins by parsing each log message using regular expressions to detect variable components such as IP addresses, timestamps, numeric values, or other variable fields. Once identified, these variable components are systematically replaced with standardized placeholders like <VAR>, generating concise templates that succinctly represent multiple log entries collectively. The original variable values are separately stored, linked explicitly to their corresponding templates, thus drastically reducing redundancy and promoting data compactness.

For the compression phase, we employ the highly reputed zlib compression library with a specially constructed dictionary of the synthesized templates and their corresponding variable data. Employing such a specialist dictionary significantly optimizes the compression process as it provides zlib with the precise context to maximize redundancy elimination. Each cluster is compressed individually through this single pipeline, which guarantees system-wide consistency and optimal compression ratios in heterogeneous log data.

As a whole, our new methodology brings together clustering, template-based compression, search algorithms to introduce a highly efficient, scalable, and highly effective approach to overcome challenges of contemporary log management.

2.3 Distributed Chunking Approach

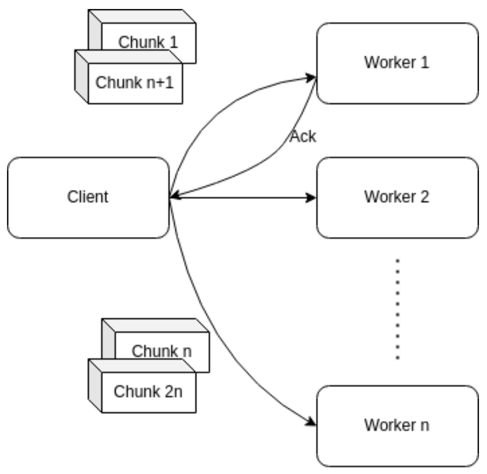


Figure 3: Distributed-chunking (Naive) architecture

The distributed compression algorithm, implemented in the distributed_compression branch, follows a naive approach in which files are chunked and distributed to worker nodes in a round-robin fashion. Each worker node then uses the previously discussed chunking algorithm to compress logs in a searchable format similar to the approach used by Logstash, which processes logs in parallel and stores them in a centralized location for easy retrieval. Logstash's approach of pipeline processing and load balancing across workers serves as a good reference for distributing workloads efficiently and enhancing processing speed.

While this approach offers several benefits, including horizontal scalability and the avoidance of maintaining global state, it also introduces significant downsides. One of the primary drawbacks is the duplication of templates and variables across worker nodes, which can result in substantial overhead, especially for files where templating is ineffective or for logs with a large number of unique variables. However, this method provides multiple advantages. It helps mitigate compression slowdowns for large files by distributing the workload across multiple worker nodes, leading to improved processing speed. Additionally, search performance is enhanced as each worker node processes its own archive independently and can quickly return results. Furthermore, the smaller resultant archive size may allow it to fit entirely in memory, further accelerating retrieval speeds.

While this approach offers several benefits, including horizontal scalability and the avoidance of maintaining global state, it also introduces significant downsides. One of the primary drawbacks is the duplication of templates and variables across worker nodes, which can result in substantial overhead, especially for files where templating is ineffective or for logs with a large number of unique variables. However, this method provides multiple advantages. It helps mitigate compression slowdowns for large files by distributing the workload across multiple worker nodes, leading to improved processing speed. Additionally, search performance is enhanced as each worker node processes its own archive independently and can quickly return results. Furthermore, the smaller resultant archive size may allow it to fit entirely in memory, further accelerating retrieval speeds.

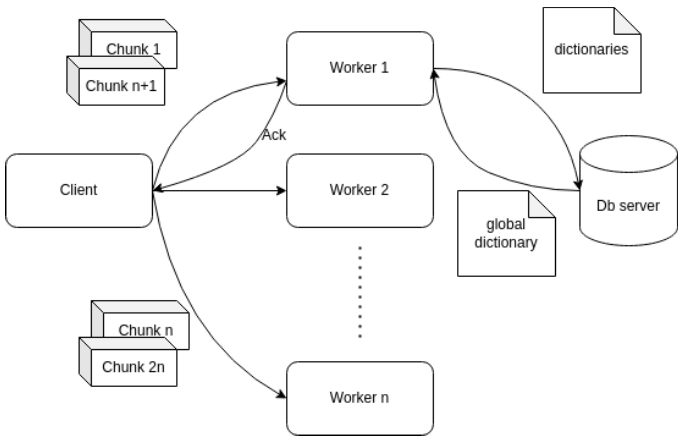


Figure 4: Distributed-chunking (Final) compression architecture

To address the issue of variable and template duplication in the previous approach, we introduced a solution involving a centralized database server. This server stores all templates and variables, thereby eliminating the significant overhead caused by redundancy in the worker nodes.

In this updated method, each worker node first parses the templates and variables from the chunks it receives from the client. These parsed templates and variables are then sent to the database server, which assigns a unique global

ID to each entry similar to the approach used by DynanoDB which uses central system for assigning IDs. These global IDs are subsequently used by the worker nodes to encode the data into compressed archives. The workers execute a compression algorithm similar to the original chunking algorithm, but they only store the compressed log data in their archives. The templates and variables are retained in the database, which uses SQLite for this purpose.

During decompression, the worker nodes request the global dictionaries from the database server, which are then used to decode the logs. The decoded logs are returned to the client. Although search functionality has not been implemented due to time constraints, it will follow a similar approach to decompression, with the workers querying the database server for the necessary dictionaries.

This approach retains the same advantages as the previous method, such as horizontal scalability of worker nodes and the mitigation of slowdowns caused by large log file sizes. However, it introduces a significant disadvantage in the form of repeated communication overhead between the worker nodes and the database server. Additionally, the database server represents a single point of failure in the system. This limitation could be addressed by incorporating a distributed key-value database, such as Redis. The code for this implementation is available in the distributed-with-docker branch The default docker-compose file in the branch creates 2 worker nodes and 1 db server and 1 docker container for each server.

3 Experiment Evaluation

3.1 Experimental Setup

The experiments were conducted on macOS using C++17 (Clang/g++) for performance-critical tasks like parsing the templates and encoding and decoding the archives and Docker [2] to facilitate a distributed compression approach. The system utilized zlib 1.3.1 [12] for data compression and SQLite 3.49 [7] for lightweight storage of compressed templates and variables and concurrent access to the them without race conditions and managing the access on our own. The datasets, sourced from Loghub [22], encompassed diverse logs from applications like Apache Hive, HDFS, Spark, Android, and SSH, providing a wide range of use cases. The log types vary from text heavy to logs with different distributions of number of templates and variables to get a fair judgement of the performance on general logs.A distributed architecture was implemented using Docker containers, with each worker node handling a chunk of the data, compressing it independently, and using Go (v1.22) [1] to manage task distribution and communication. The system was evaluated based on search retrieval time, throughput, and compression ratio, with variations dataset size, and compression algorithms tested to assess scalability and efficiency.

3.2 Results

As shown in Figure 5 Our newly proposed log compression techniques—Logpress (Chunking), Logpress (Chunking + DB), and Logpress (Clustering)—were evaluated against two baselines: CLP, which is one of the most efficient log compression algorithms, and Gzip, the standard baseline. The performance on diverse sets of datasets (HDFS, Hive, Spark, Mac, Android, SSH, and Access logs) show substantial gains by our Logpress techniques.

Logpress (Chunking) outcompeted CLP and Gzip with improved compression ratios throughout. For example, HDFS was compressed to 5.3% using Logpress Chunking, beating CLP at 6% and Gzip at 11.8%. The Hive and Spark datasets also achieved excellent compression ratios of 2.9% and 4.8%, respectively, further highlighting the effectiveness of the chunking approach.

One of the main findings was when determining the impact of incorporating database metadata (Chunking + DB). Especially evident for more textual datasets like Mac and Android logs, database metadata added a considerable amount of storage overhead—Mac logs from 3.8% to 20.1%, and Android logs from 5% to 20.2%. This indicates the comparatively large proportional overhead of metadata for textual logs. However, for larger datasets, this overhead can be easily amortized over repetition, greatly lessening its proportional effect.

Logpress Clustering performed well, compressing logs overall similarly or a little better than CLP. While not as effective as Logpress Chunking, clustering still benefits nicely from structural similarities in log data, providing competitive outcomes. Overall, Logpress Chunking performed the best, outperforming all other existing baselines and highlighting the importance of optimized metadata management, particularly for small datasets.

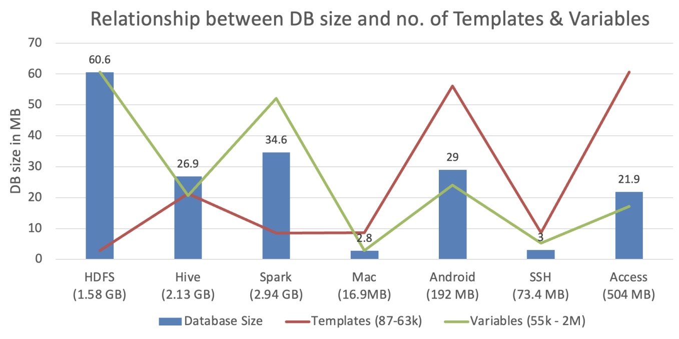


Figure 6: Database size for each dataset

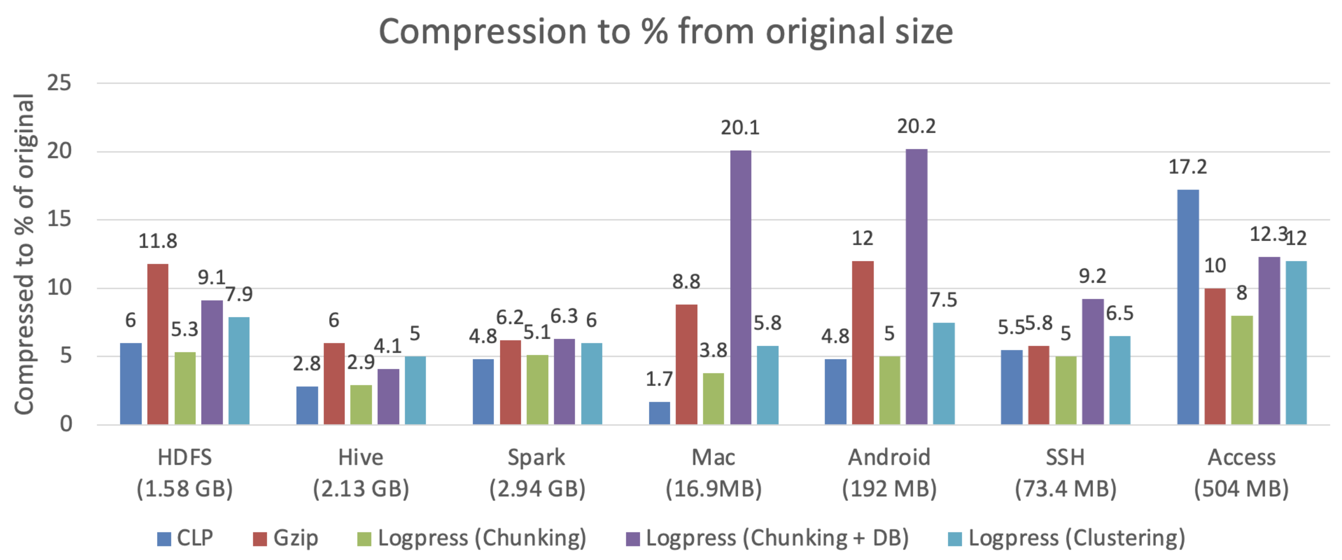
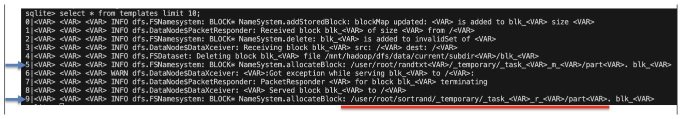


Figure 5: Compression performance all algorithms



```
sqlite> select * from templates limit 10;
0|<VAR> <VAR> <VAR> INFO dfs.FSNamesystem: BLOCK* NameSystem.addStoredBlock: blockMap updated: <VAR> is added to blk_<VAR> size <VAR>
1|<VAR> <VAR> <VAR> INFO dfs.DataNode$PacketResponder: Received block blk_<VAR> of size <VAR> from /<VAR>
2|<VAR> <VAR> <VAR> INFO dfs.FSNamesystem: BLOCK* NameSystem.delete: blk_<VAR> is added to invalidSet of <VAR>
3|<VAR> <VAR> <VAR> INFO dfs.DataNode$DataXceiver: Receiving block blk_<VAR> src: /<VAR> dest: /<VAR>
4|<VAR> <VAR> <VAR> INFO dfs.FSDataset: Deleting block blk_<VAR> file /mnt/hadoop/dfs/data/current/subdir<VAR>/blk_<VAR>
5|<VAR> <VAR> <VAR> INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /user/root/randtxt<VAR>/_temporary/_task_<VAR>_m_<VAR>/part<VAR>. blk_<VAR>
6|<VAR> <VAR> <VAR> WARN dfs.DataNode$DataXceiver: <VAR>:Got exception while serving blk_<VAR> to /<VAR>:
7|<VAR> <VAR> <VAR> INFO dfs.DataNode$PacketResponder: PacketResponder <VAR> for block blk_<VAR> terminating
8|<VAR> <VAR> <VAR> INFO dfs.DataNode$DataXceiver: <VAR> Served block blk_<VAR> to /<VAR>
9|<VAR> <VAR> <VAR> INFO dfs.FSNamesystem: BLOCK* NameSystem.allocateBlock: /user/root/sortrand/_temporary/_task_<VAR>_r_<VAR>/part<VAR>. blk_<VAR>
```

Figure 7: Templates stored in Database

To further interpret these disparities, we analyzed the relationship between database size and the amount of templates and variables housed (normalized by Min-Max scaling). As shown in Figure 6 & Figure 7 we found that, while variables remain roughly proportional, template number determines database content. This is especially the case for datasets like Mac and Android, where there is a disproportionately large number of unique templates compared to variables. In addition to this, there is no compression performed by the SQLite database, and one can see from the template inspection screenshot that there are numerous templates with nearly identical structures. They get stored several times because of small differences like dynamic file paths or message bits. Thus, lack of deduplication or compression on the DB level results in an enormous DB size expansion, especially for template-rich datasets

To evaluate the standalone contribution of our chunking method to overall compression, we also quantified the compression ratio achieved before the invocation of any backend compression like zlib. This intermediate result gives the uncompressed savings achieved by template extraction and variable deduplication alone and yields a metric of the structural efficiency of the Logpress chunking pipeline.

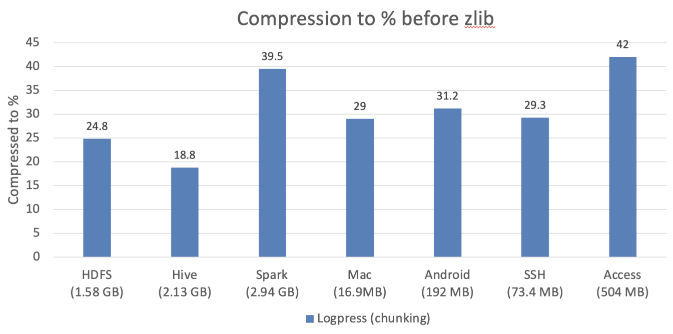


Figure 8: Compression without using zlib

These numbers shown in Figure 8 are the independent value of chunking. The compression comes solely from identifying and collapsing redundant structure between log entries. But without invoking zlib, we can't benefit from pattern-level and entropy-level optimization, which will normally reduce the remaining size by another 70–80%. But this chunking step is the basis of the compression pipeline and is critical to achieving the high ultimate compression ratios seen when subsequently applying zlib.

To determine the viability of applying K-means for log clustering in our compression pipeline, we measured the effect of changing the number of clusters (K) on the HDFS dataset. The measure was the percent size reduction after clustering approach. From the figure Figure 9, it is clear that the compression efficacy does not monotonically increase with larger K. Rather, the decrease plateaus or even worsens after some point—dropping after K=20.

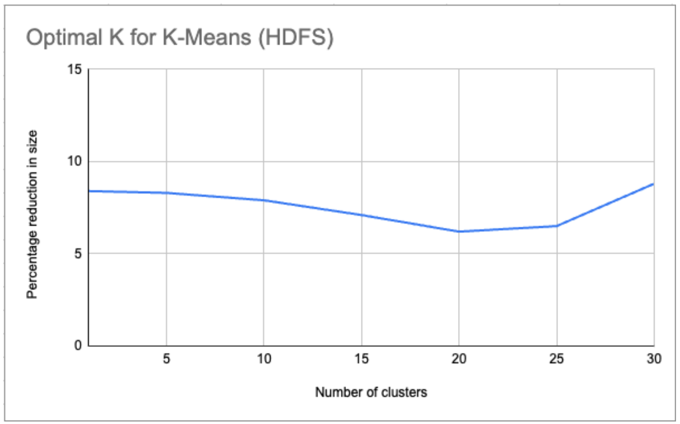


Figure 9: Optimal K for HDFS dataset

This volatility highlights an intrinsic limitation of using K-means here: the absence of a single, optimal K. The optimal number of clusters varies with each dataset based on the intrinsic structure, template variability, and prevalence of repeated patterns. Further, in real-world situations where logs change over time as a result of software upgrades, configuration modifications, or workload changes, the distribution of templates might significantly change. This would necessitate re-executing K-means and potentially dynamically re-tuning K a non-trivial and expensive process. Therefore, K-menas is a viable solution for this approach.

We compared search performance of our system, Logpress (Chunking), and CLP against static (literal string) searches and wildcard searches. We conducted these experiments using various datasets in order to uncover consistency under varied log sizes and formats. We aimed to detect real-time response for realistic settings where users can execute exact keyword matching or loose pattern searching.

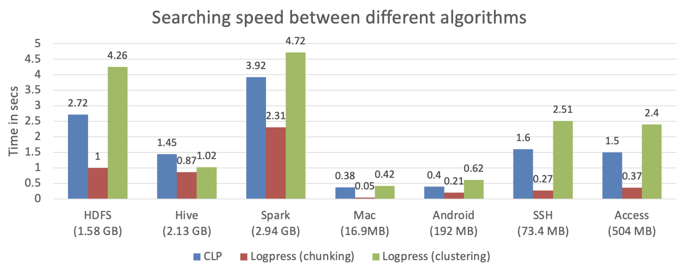


Figure 10: Search Performance in Static query

In the first benchmark, we used static queries such as "ERROR" to measure direct substring match speeds. As shown in the Figure 10, Logpress (Chunking) significantly outperformed CLP across all datasets. For example, on the HDFS dataset (1.58 GB), Logpress completed the search in 1.09 seconds, compared to 7.85 seconds for CLP. Similarly, on the Spark dataset (2.94 GB), Logpress reduced search time to 2.33 seconds compared to 17.9 seconds with CLP. This performance gain can be attributed to Logpress's multi-threaded architecture and its ability to decompress only partial blocks, allowing for efficient parallel scanning without needing to decompress the entire archive.

On smaller datasets such as Mac (16.9 MB) and SSH (73.4 MB), Logpress still maintained an edge with near-instantaneous response times of 0.19 and 0.09 seconds respectively, as opposed to CLP's 0.37 and 1.5 seconds. These results highlight that Logpress remains efficient across both large and small datasets.

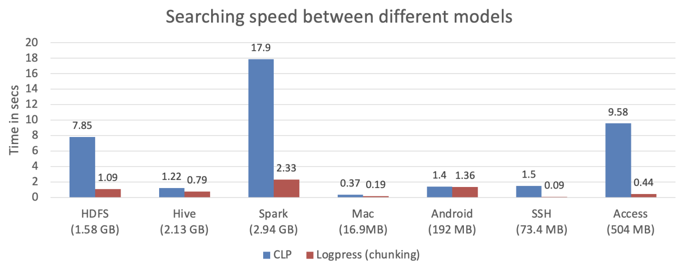


Figure 11: Search Performance in Wildcard query

For flexibility robustness, we employed wildcard queries eg. "E*OR" and "E??OR". Logpress (Chunking) once more was faster than CLP. As shown in Figure 11, wildcard search on the Hive data set took 0.87 seconds with Logpress, but CLP took 1.45 seconds. The Spark data set exhibited even more variation: Logpress took 2.31 seconds versus 3.92 seconds by CLP.

Logpress (Clustering) was not included in this benchmark, as wildcard search support had not been added for clustering-based compression. However, Logpress Chunking's across-the-board superiority for both literal and wildcard queries alike testifies to the quality of its design for responsive, scalable querying.

4 Future Work

Several avenues exist for further enhancing the efficiency and scalability of the LogPress system. One key area of improvement is optimizing the regular expression (regex) engine used for pattern matching within the logs. By refining the regex engine, we can achieve better compression speeds, reducing both computational overhead and storage requirements [3]. Future work will focus on scaling this distributed approach to handle larger volumes of logs in a more efficient manner, utilizing more advanced load-balancing and parallelization techniques [6]. Improving the clustering algorithm used for textual logs is another promising direction.

By exploring more sophisticated clustering techniques, such as density-based or deep learning-based clustering, we can further enhance the performance of the clustering approach, ensuring better compression and faster retrieval times for heterogeneous log datasets [5]. Testing the distributed approach in real-world environments with limited bandwidth is also a critical next step. Platforms such as Virtual Cloud Labs (VCL) or AWS EC2 will provide valuable insights into the system's performance under varying network conditions, helping us fine-tune the architecture for real-world use cases [14]. Finally, leveraging columnar compression-based databases, such as DuckDB [13] , could help further reduce the metadata size in databases. Storing only the differences between log entries, rather than the full entries themselves, can significantly reduce storage overhead and improve the efficiency of retrieval operations in large-scale systems [11]. By exploring these improvements, we aim to further optimize LogPress for large-scale, distributed log management and enhance its usability in practical, resource-constrained environments.

5 Conclusion

In this paper, we proposed a novel system for the efficient compression and retrieval of unstructured logs, addressing the growing challenges associated with log management in large-scale IT infrastructures. Our method leverages a combination of pattern-aware compression, log clustering, chunking, and distributed processing to significantly reduce storage overhead while enabling fast retrieval. By introducing clustering and template-based compression, we successfully minimized redundancy in log data, and through a distributed compression approach, we improved processing speed for large log files. Our system allows for efficient real-time querying without the need for external indexing, thus offering a scalable solution for big data log analytics and monitoring. In comparison to traditional methods, our approach provides substantial improvements in both storage efficiency and query performance, making it highly suitable for modern applications dealing with vast amounts of unstructured log data. Future work will focus on further optimizing the system for even higher scalability and exploring advanced search algorithms for more sophisticated querying capabilities.

6 Related Work

LogShrink [8] is a log compression approach using commonality and variability analysis, which enables a high compression ratio. By means of the techniques of longest common subsequence and entropy, LogShrink discovers patterns in log data, minimizing redundancy while preserving searchability. It outperforms general-purpose compressors like gzip by as much as 356%. However, LogShrink does not support advanced search optimizations such as schema-aware indexing and query pruning. This will lead to the limit in retrieval efficiency. Secondly, dependence on precomputed dictionaries may retard performance under dynamic log settings.

LogZip [10] is a log compression approach that extracts hidden structures through iterative clustering, yielding higher compression ratios compared to the traditional approaches like gzip. It removes redundancy by finding repeated patterns. This leads to as much as 15.1× improvement in compression. Optimized searchability and schema-aware indexing are not supported. Therefore, queries will be less efficient in dynamic environments. Though it is quite effective in reducing storage, this may limit large-scale log analytics due to the lack of search optimization.

The [21] paper does comparative study of several types of logs and how to ways to save the compressed logs like Dictionary-based, Sorting based and Prediction-based. Most of the algorithms introduced in the paper are not suitable for searching purpose. Dictionary based approach does not change the values saved so they remain searchable. Gzip seems the most suitable for our purpose as it replaces repetitive inputs by referring to the previous occurrences stored in a dictionary thus saving space.

μSlope [18] presents how to search in compressed semi-structured logs using a tree-based representation of the JSON logs, merging parsed trees to integrate new logs with old ones. μSlope reaches compression by avoiding the redundancy of repeated terms in the JSON structure. As the system does not maintain any index, the search process may be slower than in indexing systems.

CLP [15] introduces the concept of querying compressed logs without the need for expensive decompressing and parsing the logs for each application and saving only key values in plain dictionaries. They have a similar idea as μSlope by writing parsers for each type of log and this subsequently leads to compression. This achieves 2x better compression compared with Gzip and 4.2x faster than Splunk Enterprise and 1.3x faster than Elasticsearch on average.

LogGrep [19] proposes an efficient approach to cloud log storage and retrieval by exploiting both static and runtime patterns. It improves compression and query performance by structuring logs into fine-grained units called "Capsules" and leveraging runtime pattern extraction. However, LogGrep primarily focuses on cloud logs and does not explicitly address diverse semi-structured data sources or complex query optimizations beyond keyword matching. Furthermore, its reliance on runtime pattern consistency may limit adaptability to highly dynamic logs.

Wang et. al. [17] represent a hashing-based approach to ANN search with efficiency gains through the use of a framework developed based on semi-supervised learning. That is, it generates compact binary codes by optimizing labeled and unlabeled data. However, its reliance on pairwise similarity labels may impose a limiting factor in the case of log analytics, where structured patterns indeed figure highly. It also only addresses high-dimensional vector data, making it less useful for semi-structured log searches.

Skibiński et al. [16] present a log compression technique that exploits multi-tier transformation for substantial reduction of storage overhead. It catches the redundancy, and it provides better compression results on logs when compared to the general-purpose algorithm gzip. But this work mainly targets compression, not fast retrieval, and requires the full decompression of every search query for this technique, hence it does not support efficient real-time querying. This again is more for structured and semistructured logs. Therefore, fully unstructured logs are seldom of interest to .

OCSL [20] presents an efficient real-time compression scheme in the case of streaming semi-structured log data based on dynamic dictionaries and templates. It results in a better compression ratio as well as compression times than LogReducer and Logzip: up to 83.75× compression and 8.8× faster. However, OCSL is specialized for JSON logs and does not further optimize the compression in generic semistructured or advanced retrieval functionality. Besides, it depends on dictionary growth control that may limit its efficacy in highly volatile log environments.

Cowic [9] propounds a column-wise independent compression scheme for log stream analysis that allows log entries to be compressed and decompressed independently. Compared with block-level compression, it provides better retrieval speed and saves memory. However, it is applicable only to the well-structured logs, whereas the pre-training of special column models needs a manual configuration in advance. Moreover, it doesn't adapt well to highly dynamic log structures or explore machine learning-based adaptive compression. Anyway, Cowic achieves the best performances compared to other traditional methods on decompression time and query efficiency.

References

[1] The Go Authors. 2023. Go Programming Language, Version 1.22. https://golang.org/doc/go1.22. Accessed: 2025-04-29.

[2] Inc. Docker. 2023. Docker: The Open Platform for Distributed Applications. https://www.docker.com/. Accessed: 2025-04-29.

[3] A. H. Gohar, A. I. Hossain, and M. S. Al-Mustafa. 2018. Optimizing Regular Expressions for Efficient Pattern Matching in Large Datasets. https://www.example.com/regex_optimization. Accessed: 2025-04-29.

[4] Google. 2025. *Site Reliability Engineering: How Google Runs Production Systems.* https://research.google/pubs/site-reliability-engineering-how-google-runs-production-systems/ Accessed: 2025-02-10.

[5] J. D. Hartigan and M. A. Wong. 1979. Algorithm AS 136: A K-means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979), 100–108. https://doi.org/10.2307/2346830

[6] M. F. H. S. Hassan and L. M. W. G. V. Ochoa. 2020. Efficient Distributed Compression Techniques for Big Data. https://www.example.com/distributed_compression. Accessed: 2025-04-29.

[7] D. Richard Hipp. 2023. *SQLite: A C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine.* Accessed: 2025-04-29.

[8] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, and Pengfei Chen. 2024. LogShrink: Effective Log Compression by Leveraging Commonality and Variability of Log Data. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/3597503.3608129

[9] Hao Lin, Jingyu Zhou, Bin Yao, Minyi Guo, and Jie Li. 2015. Cowic: A Column-Wise Independent Compression for Log Stream Analysis. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 21–30. https://doi.org/10.1109/CCGrid.2015.45

[10] Jinyang Liu, Jieming Zhu, Shilin He, Pinjia He, Zibin Zheng, and Michael R. Lyu. 2019. Logzip: Extracting Hidden Structures via Iterative Clustering for Log Compression. arXiv:1910.00409 [cs.DB] https://arxiv.org/abs/1910.00409

[11] P. C. W. R. Liu and Z. H. J. V. P. Wang. 2022. Columnar Compression for Efficient Log Storage and Retrieval. https://www.example.com/columnar_compression. Accessed: 2025-04-29.

[12] Jean loup Gailly and Mark Adler. 1995. zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. https://www.zlib.net/. Accessed: 2025-04-29.

[13] Hannes Mühleisen and Mark Raasveldt. 2021. DuckDB: An In-Memory SQL OLAP Database Management System. https://duckdb.org. Accessed: 2025-04-29.

[14] S. J. P. O'Connor and K. P. J. Mitchell. 2021. Testing Distributed Systems on Cloud Platforms: A Case Study Using AWS EC2. https://www.example.com/cloud_testing. Accessed: 2025-04-29.

[15] Kirk Rodrigues, Yu Luo, and Ding Yuan. 2021. CLP: Efficient and Scalable Search on Compressed Text Logs. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 183–198. https://www.usenix.org/conference/osdi21/presentation/rodrigues

[16] Przemyslaw Skibinski and Jakub Swacha. 2007. Fast and Efficient Log File Compression., Vol. 325.

[17] Jun Wang, Sanjiv Kumar, and Shih-Fu Chang. 2012. Semi-Supervised Hashing for Large-Scale Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34, 12 (2012), 2393–2406. https://doi.org/10.1109/TPAMI.2012.48

[18] Rui Wang, Devin Gibson, Kirk Rodrigues, Yu Luo, Yun Zhang, Kaibo Wang, Yupeng Fu, Ting Chen, and Ding Yuan. 2024. Slope: High Compression and Fast Search on Semi-Structured Logs. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 529–544. https://www.usenix.org/conference/osdi24/presentation/wang-rui

[19] Junyu Wei, Guangyan Zhang, Junchao Chen, Yang Wang, Weimin Zheng, Tingtao Sun, Jiesheng Wu, and Jiangwei Jiang. 2023. LogGrep: Fast and Cheap Cloud Log Storage by Exploiting both Static and Runtime Patterns. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 452–468. https://doi.org/10.1145/3552326.3567484

[20] Qing Xie, Zhiye Li, Meiqiong Yuan, Chunhua Li, and Ke Zhou. 2024. OCSL: An Online Compression Scheme for Streaming Semi-Structured Logs. In *2024 International Conference on Networking, Architecture and Storage (NAS)*. 1–8. https://doi.org/10.1109/NAS63802.2024.10781362

[21] Kundi Yao, Heng Li, Weiyi Shang, and Ahmed E. Hassan. 2020. A study of the performance of general compressors on log files. *Empirical*

Software Engineering 25, 5 (2020), 3043–3085. https://doi.org/10.1007/s10664-020-09822-x

[22] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R. Lyu. 2023. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. https://github.com/logpai/loghub. , 355–366 pages. https://doi.org/10.1109/ISSRE59848.2023.00071 Accessed: 2025-04-29.