LogCloud: Fast Search of Compressed Logs on Object Storage

Ziheng Wang
Stanford University
zihengw@stanford.edu

Junyu Wei
Tsinghua University
thuweijy@vip.163.com

Alex Aiken
Stanford University
aiken@cs.stanford.edu

Guangyan Zhang
Tsinghua University
gyzh@tsinghua.edu.cn

Jacob O. Tørring
NTNU
jacob.torring@ntnu.no

Rain Jiang
Bytedance
rain.jiang@bytedance.com

Chenyu Jiang
Bytedance
chenyu.jiang@bytedance.com

Wei Xu
Bytedance
wei.xu@bytedance.com

ABSTRACT

Large organizations emit terabytes of logs every day in their cloud environment. Efficient data science on these logs via text search is crucial for gleaning operational insights and debugging production outages. Current log management systems either perform full-text indexing on a cluster of dedicated servers to provide efficient search at the expense of high storage cost, or store unindexed compressed logs on object storage at the expense of high search cost.

We propose LogCloud, a new object-storage based log management system that supports both cheap compressed log storage and efficient search. LogCloud constructs inverted indices on compressed logs using a novel FM-index implementation that supports efficient querying from object storage directly, removing the need for dedicated indexing servers. Experiments on five public and five production log datasets show that LogCloud can achieve both cheap storage and search, scaling to TB-scale datasets.

PVLDB Reference Format:
Ziheng Wang, Junyu Wei, Alex Aiken, Guangyan Zhang, Jacob O. Tørring, Rain Jiang, Chenyu Jiang, and Wei Xu. LogCloud: Fast Search of Compressed Logs on Object Storage. PVLDB, 18(8): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/marsupialtail/rottnest-vldb-repro.

1 INTRODUCTION

Modern organizations' cloud infrastructure generate terabytes of logs every day. These logs are typically stored in a log management system like ElasticSearch, DataDog or Splunk and queried interactively for troubleshooting or cybersecurity use cases [15, 19, 47]. Organizations would like to do both of the following:

- **Cheap Retrieval:** It must be possible to cheaply and interactively search these logs with wildcard string queries (e.g. *.amazon.*, pod-abcd-*).
- **Cheap Storage:** The cost of storing the logs and associated index structures to support cheap search must also be low.

Current approaches to log management represent two extremes in the trade-off between storage and compute costs. Traditional solutions like Splunk, Datadog and OpenSearch maintain dedicated servers that are always running, coupling compute and storage. While these systems provide fast query performance through full-text search indices kept "hot" in memory/disk, they require organizations to continuously pay for expensive compute resources regardless of actual query volume, making them cost inefficient in cases where the logs are less frequently accessed [15, 39, 47, 52].

The alternative approach, adopted by systems like Grafana Loki, Datadog Flex logs, or simply storing and querying compressed files on object storage [16, 25, 45, 52], decouples compute and storage by eliminating always-on servers and storing unindexed compressed logs. However, without index structures, these systems must resort to brute-force scanning, which becomes expensive at scale - e.g. Datadog's new Flex log offering can cost tens of thousands in reserved compute capacity [49]. While this serverless approach works for sporadically queried audit logs, the per-scan compute costs quickly make it more expensive than traditional solutions for frequently accessed logs.

We show that object storage-native full-text search indices solve the high query cost problem in compute-storage decoupled log management systems. Drawing on recent advances in data lake indexing [41, 54], we build LogCloud, which maintains complete storage-compute separation through indices optimized for efficient cold access, eliminating always-on servers while reducing query costs. This positions LogCloud as the most cost-effective solution across a large range of intermediate query loads where integrated systems' high upfront costs are not justified and disaggregated systems' high per-query costs become inefficient.

LogCloud first uses a state-of-the-art log compression system, LogGrep, to drastically reduce the amount of text that has to be indexed by breaking logs down into template and variable components (e.g. *unique resource identifiers* (URI) like kube pod names) [52]. LogCloud then constructs inverted indices on the variables. The main technical challenge addressed in LogCloud is designing an inverted index that supports **efficient substring search on object storage with minimal storage overhead**. We address this challenge using an FM-index based on the *Burrows Wheeler Transform (BWT).* While FM-indices have been extensively studied for disk/RAM settings [2, 11, 23, 24, 27], their traditional implementations are poorly suited for object storage due to its high access latency. We propose a novel implementation specifically optimized for object storage that significantly improves search latency while maintaining high compression ratios.

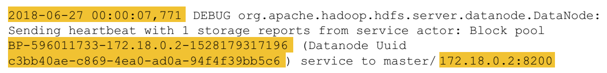
In summary, this paper makes three key contributions:

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:XX.XX/XXX.XX

- Propose object-storage-native indices as a solution to the high query cost problem for compute-storage decoupled log management systems by avoiding scanning the entire dataset.
- Address latency and size challenges in the inverted index through a novel FM index and suffix array optimized for object storage.
- Experimentally demonstrate that LogCloud enables interactive search over TB-scale compressed logs on object storage in under 10 seconds on a single machine, achieving comparable or better performance than dedicated log search services like OpenSearch UltraWarm with less than 10% of the storage footprint on five public log datasets. This translates to a >10x *total cost of ownership* (TCO) saving for large log datasets on over four orders of magnitude of total query load at a 12-months operating horizon.

2 MOTIVATION

A full-text index for compressed logs that supports efficient querying directly from object-storage is key to LogCloud's goal of lowering query cost while maintaining compute-storage decoupling. While attempting to adapt prior full-text indexing and log compression techniques to an object storage environment, we encountered significant challenges due to its high read latency. These challenges led us to develop novel adaptations of the FM-index and suffix array specifically optimized for object storage.



```
2018-06-27 00:00:07,771 DEBUG org.apache.hadoop.hdfs.server.datanode.DataNode:
Sending heartbeat with 1 storage reports from service actor: Block pool
BP-596011733-172.18.0.2-1528179317196 (Datanode Uuid
c3bb40ae-c869-4ea0-ad0a-94f4f39bb5c6 ) service to master/172.18.0.2:8200
```

Figure 1: Logs are typically made up of fixed *templates* and changing *variables*, which are highlighted in yellow. Logs that do not fit into common templates are called *outliers*.

2.1 Background: Inverted Indices after Log Compression

Recent work like CLP and LogGrep has demonstrated that logs are highly repetitive and can be effectively compressed by exploiting static and runtime patterns, as shown in Figure 1 [44, 52, 53]. Almost all logs can be decomposed into repeated *templates* and *variable* components (e.g., request IDs or pod names, typically long pseudo-random alphanumeric URIs). LogCloud uses LogGrep to first decompose logs and indexes the variable components only. The templates, typically small in size, can be brute force searched.

To build the index itself, we leverage an inverted index structure that maps each variable to a *posting list* - an ordered collection of document IDs and positions where the token appears. These tokens are stored in a *term dictionary*, a collection of all unique tokens with pointers to their posting lists. The challenge lies in efficiently managing the *secondary index* needed to quickly look up tokens in this term dictionary, which can grow to multiple GBs when dealing with URI-style variables. Two popular approaches for this secondary index are finite state transducers (FST), used by ElasticSearch, OpenSearch, and M3DB [20, 34, 36, 39], and sorted string tables (SSTables) [40], adopted by systems like Cassandra and Quickwit [17, 32, 43].

2.2 Challenge: Substring Searches on URIs

While both FSTs and SSTables enable efficient prefix (query∗) and exact-match string queries on these variable tokens, they lack support for efficient substring (∗query∗) searches. Some systems (Quickwit, Cassandra) simply do not support substring search, while others (ElasticSearch) cannot efficiently use the secondary index and perform expensive scans of the term dictionary [20, 32, 43].

In search engine use cases, a term dictionary scan is acceptable, as the size of the language vocabulary does not grow linearly with the amount of text being indexed. However, as shown in Figure 1, the term dictionary here consists of unique resource identifiers (URIs) whose number increases linearly with the size of logs being indexed. We will show in Section 5.2 that scanning this term dictionary can be very expensive for larger datasets.

Substring searches are critical for observability and cybersecurity use cases [9, 30, 46]. For example, an engineer troubleshooting a service outage might search for a partial URI '172.18.0.2' embedded in a larger URI, as shown in Figure 1, to correlate across log sources. As a second example, a security analyst investigating potential threats needs to search for partial IP addresses or domain fragments in network logs to identify suspicious traffic patterns (e.g., searching for "10.0.0." to find all matching IPs, or ".xyz" to detect traffic to suspicious top-level domains). In addition to these practical use cases, users often rely on substring queries rather than prefix or exact matches to ensure comprehensive results, particularly when the log management framework's tokenization scheme is unfamiliar and missing matches is unacceptable.

In the authors' experience operating large-scale distributed systems in industry, substring queries dominate incident response workloads to debug failures and detect intrusions. Efficient support for substring queries is thus a basic requirement for LogCloud.

2.3 Solution: The BWT and FM-index

What object-storage based secondary index would allow efficient substring searches on the term dictionary? Apart from FSTs and SSTables, two other full-text indexing approaches have been proposed in literature. The first are grammar-based compression approaches like Sequitur [10, 12, 38, 55, 56] and the second are succinct data structures like the Burrows Wheeler Transform (BWT) [2, 11, 23, 33]. We choose the second approach in LogCloud for two reasons. First, grammar-based approaches heavily rely on repeated subwords that occur frequently in natural language text but rarely occurs in the URIs that we are indexing. Second, while the compression costs of Sequitur-based algorithms can be prohibitively high, efficient industrial-grade implementations for performing the BWT exist [33, 37, 55].

LogCloud uses the FM-index based on the BWT, an example of which is shown in Figure 2. The FM-index is a common data structure typically used in bioinformatics to perform substring searches in DNA read mapping. To obtain the BWT of an input text corpus (the term dictionary in our case), generate a matrix of cyclic permutations of the corpus, i.e. all the rotations of BANANA in the example. Then, these permutations are sorted lexicographically. The last column from the array of suffixes, highlighted in the red box, is called the BWT [22].

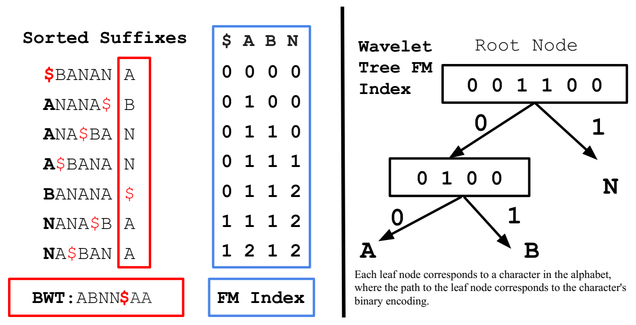


Figure 2: Summary of BWT and FM-index on the input string BANANA. For a more illustrated reference see [50]. We show a simple FM Index and a wavelet tree FM-index. To compute $rank(B, 4)$ with a wavelet tree, we first lookup B's binary representation 01. Since the first digit is 0, we find $rank(0, 4) = 2$ in the bitvector at the root node. Then we go down the left branch and find $rank(1, 2) = 1$ as the result.

Algorithm 1 Iterative Substring Search using FM-index with BWT

```
1:  procedure FM_SEARCH($P, BWT$)  ▷ $P$ is the substring to search
2:      $l \leftarrow 0, r \leftarrow |BWT|$
3:      $C \leftarrow$ counts of each character in $BWT$
4:      for $i \leftarrow |P|$ down to 1 do
5:          $l \leftarrow C[P[i]] + rank(P[i], l)$
6:          $r \leftarrow C[P[i]] + rank(P[i], r)$
7:          if $l \geq r$ then return "Pattern not found"
8:          end if
9:      end for
10:     return Pattern found between BWT positions $l$ and $r$
11: end procedure
```

The BWT is used to construct the FM-index, which allows efficient substring searches. The FM-index enables efficient computation of $rank(c, i)$, defined as *how many times character c has appeared up to position i in the BWT*. Assuming $rank(c, i)$ can be efficiently computed for all characters in the alphabet and all positions in the BWT, Algorithm 1 is commonly used to find all occurrences of a substring P in the input text corpus using the rank operation repeatedly [23]. Figure 2 shows the simplest FM-index, which just records this number for all c and all positions.

The FM-index is typically implemented with a *wavelet tree* in RAM or disk-based used cases [27, 31, 35]. The wavelet tree compresses the BWT into a binary tree, where each node contains a bitvector. To retrieve the rank of a character, the tree is traversed from the root with rank operations done on the bitvectors at each node. A tree traversal for the BWT "ABNNAA" is in Figure 2.

The result of Algorithm 1 indicates the query pattern is found between positions l and r of the BWT, which needs to be mapped back to locations in the original text corpus. This can be done very quickly with a list that records the offset in the original corpus that corresponds to each position in the BWT, called the *suffix array*. However, this is a list of integers as long as the original text corpus, and is in general very poorly compressible. A common technique used in literature is the *sampled suffix array*, which stores only offsets for every K positions. If a position i's offset is not stored, the FM-index has to be repeatedly consulted to relate position i's offset to $i - 1$'s offset until a sampled location is hit [2, 23].

2.4 Challenge: Query Latency on Object Storage

To the best of our knowledge, all existing implementations of the FM-index have targeted disk or in-memory scenarios. This is because the FM-index is typically used to map short reads against a reference genome, which rarely exceeds several GBs in size. However, in our scenario, we would like the index to reside on object storage, which has a higher read latency of tens of milliseconds [18]. This raises two critical challenges for the standard wavelet tree FM-index implementation.

The first challenge is the latency of substring search with the wavelet tree. In a wavelet tree, each rank operation takes $O(H_C)$ sequential random reads, where H_C denotes the entropy of the alphabet. Since we are constructing the index on pseudorandom variables like URIs, the entropy is the log of the size of the alphabet. Thus for alphanumeric variables, around six sequential reads to object storage are required to compute one rank operation with the wavelet tree. Algorithm 1 shows that we compute $|P|$ rank operations sequentially. Long queries such as 'nginx-554b9c67f9-c5cv4' can require tens of rank operations, which translates to hundreds of sequential read requests to the object storage.

The second challenge relates to the latency of accessing the sampled suffix array used to map BWT positions back to locations in the input text. While accessing the FM-index up to K times for each mapped BWT position can be acceptable when the FM-index is in memory or on disk, it incurs unacceptable latency for object storage. This is particularly problematic as hundreds of positions potentially have to be mapped. Even though querying each position can be parallelized, making thousands of small concurrent requests to object storage may run into S3 request throttling [3]. Alternatively, one could opt to store the full suffix array, but it has a very high storage footprint, which would annul the benefits we obtain from log compression [23].

3 OBJECT STORE NATIVE INVERTED INDEX

LogCloud effectively tackles the two challenges by focusing on the **IO-bound** and **latency-bound** nature of object storage, where data retrieval is significantly more expensive compared to processing the data and retrieving 1 byte and 1 MB have similar latency.

Based on these observations, we introduce two key innovations: (1) a custom object storage-optimized FM-index that reduces sequential requests for substring queries from $O(H_C|P|)$ to $O(|P|)$, and (2) a range-reduced full suffix array approach that maintains performance while drastically reducing storage requirements through effective compression. Together, these innovations adapt the FM-index and suffix array to address the challenges of efficient log search on object storage.

3.1 Fast Search with Custom FM-Index

We tackle the first challenge through a novel object-storage-optimized implementation of the FM-index, reducing the sequential requests for a substring query of length P from $O(H_C|P|)$ to $O(|P|)$ versus the standard wavelet tree implementation. The BWT is divided into fixed-size chunks, and we compress each chunk and store the rank of every character in the BWT up to the beginning of the chunk in each chunk. The details are in Algorithm 2. The built chunks can be stored contiguously on object storage together with an offsets array that contains the byte range of each chunk. Algorithm 3 can then be used to compute $rank(c, i)$.

Algorithm 2 Build Chunks for Custom FM Index

```
1: function BuildChunks($BWT, chunk\_size(cs) = 4M$)
2:     $chunks \leftarrow [], ranks \leftarrow \{c : 0 \text{ for } c \in \Sigma\}$
3:     for $i \leftarrow 0$ to $\lceil |BWT|/cs \rceil - 1$ do
4:         $chunk \leftarrow (compress(BWT[i \cdot cs : (i+1) \cdot cs]), ranks)$
5:         $ranks[c] \leftarrow ranks[c] + count(c, BWT[i \cdot cs : (i+1) \cdot cs])$ for $c \in \Sigma$ ▷ Update global ranks with counts in this chunk.
6:         $chunks.append(chunk)$
7:     end for
8:     return $chunks$
9: end function
```

Algorithm 3 Rank Computation using Custom FM Index

```
1: function Rank($c, i$, chunks, chunk_size (cs))
2:     $chunk \leftarrow chunks[\lfloor i/cs \rfloor]$ ▷ Locate chunk containing pos i
3:     $text, ranks \leftarrow chunk$ ▷ Chunk contains compressed BWT and the ranks of each character up to the start of the chunk
4:     $local\_pos \leftarrow i \bmod cs$
5:     $decompressed \leftarrow$ decompress($text$)
6:     $local\_count \leftarrow 0$ ▷ Compute rank of $c$ in this chunk.
7:     for $j \leftarrow 0$ to $local\_pos$ do
8:         if $decompressed[j] = c$ then
9:             $local\_count \leftarrow local\_count + 1$
10:        end if
11:    end for
12:    return $ranks[c] + local\_count$ ▷ Final rank = rank up to this chunk + local rank.
13: end function
```

This approach requires reading just one chunk to compute the rank and is much simpler than the wavelet tree design. This implementation, inspired by the original FM-index implementation based on the occurrences matrix and Jacobson's rank [22, 29], is not popular for typical disk/RAM-based FM-index implementations because the rank calculation within the chunk is now done on characters, which is much more compute-intensive than rank calculations on bits that have hardware acceleration like popcnt instructions. However, in our IO-bound scenario this computation cost is easily eclipsed by the read cost.

Another reason why this approach is not typically preferred is because uncompressed, it takes around the same space as the input corpus. The wavelet tree representation comes with native compression as the storage footprint of each character is the size of its binary encoding (e.g. Huffman code). However, we can compress each character chunk in our FM-index using generic compression like Zstd [21] and decompress the chunk upon reading. Decompression adds too much overhead for disk/RAM-based FM-indices since reading is fast, but again acceptable in our IO-bound case: decompressing a chunk in memory is much faster than downloading the chunk from object storage. For example, downloading 512 300KB Zstd compressed chunks from S3 in parallel is only 5% slower than downloading and decompressing them concurrently, compared to 70% slower from NVMe SSD on an r6id.2xlarge instance on AWS.

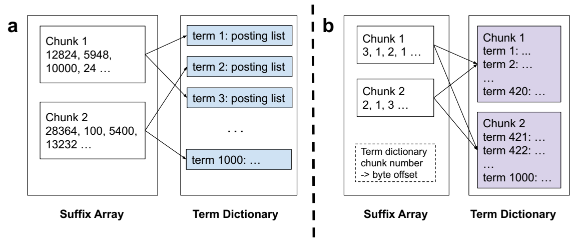


Figure 3: Range reduction to compress the suffix array.

3.2 Full Suffix Array with Range Reduction

We resolve the second challenge by storing a heavily compressed full suffix array instead of a sampled suffix array. The FM-index points us to positions l and r in the BWT. We rely on the suffix array to map these positions back to offsets in the term dictionary. As discussed in Section 2.3, the suffix array contains as many 64 bit integers as characters in the term dictionary, whose massive size can negate any of our log compression benefits.

Similar to the FM-index, we store the full suffix array in chunks, and compress each chunk. To fetch positions l to r, the chunks containing those positions are downloaded and filtered for these positions. However, if the chunks contain byte offsets of the posting lists in the term dictionary (Figure 3a), they are still very poorly compressible because they would contain a wide range of large integers with minimal patterns or repetition, making standard compression algorithms like Zstd ineffective at reducing their size.

In LogCloud, instead of storing offsets into the original term dictionary, we break the term dictionary into chunks, and only record the chunk number in the suffix array (Figure 3b). Even though we still have to store the same number of integers as the naive approach, the dynamic range of each integer is reduced by several orders of magnitude. Subsequent positions in the suffix array are also now more likely to be identical. This makes generic compression like Zstd very effective on the suffix array. We call this optimization technique ***range reduction***.

This optimization is motivated by the observation that byte-range GET requests on object storage up to around 1MB are all latency bound and have roughly the same speed. As a result, it is unnecessary for the secondary index to point us to the exact 20-byte term in the term dictionary. It is sufficient to point to the 1MB chunk that contains the term, then download and scan the chunk exhaustively to locate the term. The scan cost is usually insignificant compared to the download.

4 LOGCLOUD ARCHITECTURE

We now discuss how the novel object-storage native inverted index fits in the overall architecture of LogCloud. LogCloud consists of two key components, indexing and querying, shown in Figure 4.

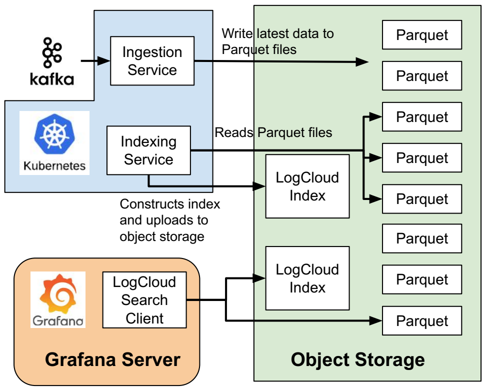


Figure 4: LogCloud's architecture.

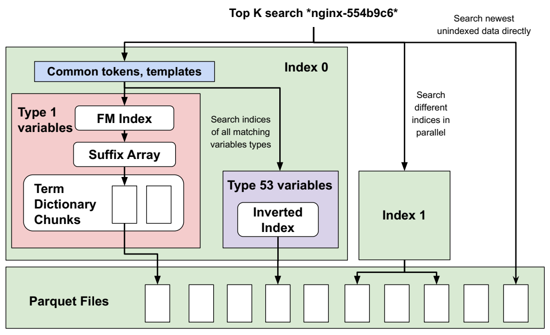

Top K search *nginx-554b9c6*
Search newest unindexed data directly
Common tokens, templates
Index 0
Type 1 variables
FM Index
Suffix Array
Term Dictionary Chunks
Search indices of all matching variables types
Type 53 variables
Inverted Index
Search different indices in parallel
Index 1
Parquet Files

Figure 5: Searching workflow in LogCloud. All data structures shown in boxes are stored on object storage.

4.1 Indexing

Similar to other compute-storage decoupled log management systems, LogCloud runs an ingestion pipeline that dumps logs in Parquet format on object storage. What sets LogCloud apart from other such systems is that it also runs an indexing service. Once a configurable amount of new logs have been collected, a LogCloud inverted index is built on the new data. We select Parquet to store the raw logs due to its mature support for compression and other analytics engines like SparkSQL, which can be used to supplement LogCloud. LogCloud can also directly index logs in Parquet ingested by another system, e.g. AWS Security Lake [6, 13, 45].

During indexing, we first use LogGrep [52] to break down ingested logs into template and variable components, categorizing variables into 64 *types* based on their character composition (e.g. only numeric, alphanumeric etc) [52]. For each type, LogCloud builds an inverted index with a term dictionary divided into 1MB chunks, as described in Section 3.2. The posting list points to Parquet pages, which are chunks of a few hundred KBs of compressed data. We find that we can download and search hundreds of Parquet pages in parallel in hundreds of milliseconds from an EC2 instance with a heavily optimized custom Parquet reader in Rust.

If the compressed term dictionary exceeds 5MB, we construct the secondary FM index and suffix array described in Section 3.1 to efficiently look up term dictionary chunk numbers from substring queries. Otherwise, we simply scan the term dictionary. Altogether, a LogCloud index file contains the templates, term dictionary chunks, and optionally the FM index and suffix array.

4.2 Searching

LogCloud provides an embedded client library to search its index and Parquet files on object storage. This offers flexible deployment options anywhere that can access the object storage bucket containing these files, such as on a Grafana server or on a serverless function. The search functionality operates completely independently from indexing and requires no always-on servers.

The search process for a top-K substring query is illustrated in Figure 5. The latest unindexed data is scanned in Parquet directly. LogCloud queries all the built LogCloud indices in parallel. To query each index, the following steps occur:

- The extracted templates are downloaded from object storage and searched exhaustively. If the substring query matches here, the searcher will simply abort using the index and brute force search all the Parquet files since the target substring occurs frequently.
- URI substring searches, e.g. "∗55493∗", will not match common templates, leading the searcher to search the inverted indices for the variables, described in Section 3. The LogGrep type of the query is determined and the inverted indices for all "compatible" types are searched in parallel. A compatible type is a type that could contain the type of the query. For example, if the query contains only numbers (type 1 in LogGrep), the type containing all alphanumerics also must be searched (type 53).
- The search client first queries our custom FM-index to find positions l and r in the suffix array as described in Algorithm 1, then retrieves term dictionary chunk numbers from the suffix array between these positions. The chunks are downloaded and regex searched for matches, with any matches leading to retrieval and search of the referenced Parquet file pages.

Certain parts of the index, like the templates and FM-index metadata, are small and accessed repeatedly across queries. While these characteristics make them ideal candidates for client-side disk caching, we do not explore this in our evaluation to maintain a straightforward comparison with other systems due to the high variability in cache-hit rates across various log analytics use cases.

5 RESULTS

We compare LogCloud against two representative baseline systems: OpenSearch UltraWarm, which exemplifies compute-storage integrated indices, and LogGrep, which represents the compute-storage disaggregated approach of downloading and scanning compressed logs [5, 52]. For the LogGrep baseline, we compress the logs using LogGrep and store them in object storage. During search, the compressed logs are downloaded and searched on NVMe SSD.

We use four LogHub datasets, HDFS (1.5GB), Thunderbird (30GB), Hadoop (17GB), Windows (26GB) [52, 57], as well as a 429GB dataset named Cluster from [44]. For each dataset, we test three search queries: common keyword, exact-match URI, and substring URI, returning top 1000 results. For example, on the Hadoop dataset, we search for 'blk_1076115144∗', '∗1076115144∗' and ERROR.

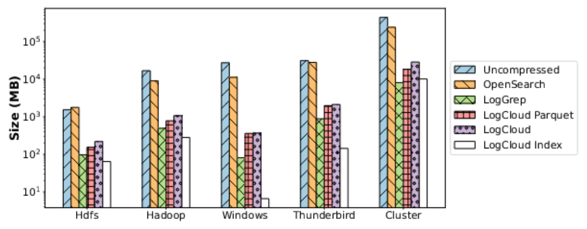


Figure 6: Storage footprint comparisons across five datasets.

We run LogCloud and LogGrep searcher on a single r6i.xlarge EC2 instance (4 vCPUs, 32GB RAM) [51], with LogCloud indices, Parquet files and LogGrep compressed files stored on AWS S3 in the same region. AWS OpenSearch UltraWarm uses three r7g.large nodes (2vCPU, 16GB RAM) and three ultrawarm1.medium instances (2 vCPUs, 15.25GB RAM). All measurements are repeated five times, with the standard deviation shown where applicable.

5.1 Storage Footprint

First, we compare the storage footprint of the different log management solutions in Figure 6. For LogCloud we show both the size of just the Parquet files and the total size with the index files. Across all five datasets, LogCloud (Parquet + index) achieves 11.8x geomean lower storage footprint against OpenSearch and 2.8x larger storage footprint compared to LogGrep. The LogCloud index itself achieves 93x geomean lower storage footprint compared to OpenSearch and 2.8x lower compared to LogGrep. We make the following observations:

- Consistent with prior findings [44, 52], OpenSearch exhibits poor space efficiency, with its index size approaching that of the raw uncompressed logs. Moreover, OpenSearch UltraWarm incurs additional operational costs due to its requirement for continuously running servers, described more in Section 5.3.
- For most log types, LogCloud storage size is dominated by the Parquet files. As expected, LogGrep's storage footprint is smaller since its log-specific compression outperforms the Zstd compression used in Parquet [52].

The second observation raises the question if we can further improve the storage footprint by moving away from Parquet: instead of Parquet's zstd compression, we could use LogGrep to compress chunks of logs and have LogCloud's posting lists point to those chunks. However, this would sacrifice crucial interoperability with external SQL engines and data lakes [1, 7, 45].

5.2 Search Latency

In Figure 7 we show the search performance of the three different query types on the five log datasets with OpenSearch, LogGrep and LogCloud. We break down the LogGrep runtime into the time it takes to download the compressed logs and the time to search the downloaded files on disk. We break down the LogCloud runtime into searching the index on object storage and downloading and filtering the matched Parquet pages.

On the queries on common tokens such as "ERROR" that match log templates, LogCloud sidesteps the inverted index and directly searches Parquet files, spending almost no time in index search.

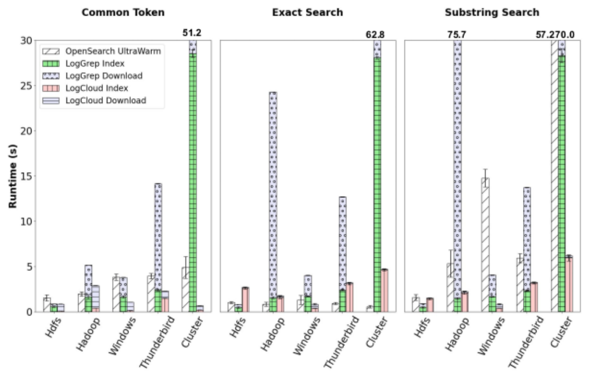


Figure 7: Search times for different query types across five public log datasets with breakdowns between search and download for LogGrep and index and Parquet for LogCloud. Bars exceeding the y-axis are annotated.

This leads to a 2.2x geomean speedup over OpenSearch and 5.0x over LogGrep, consistent over almost all log types.

For URI queries, LogCloud's index search dominates query time over Parquet page retrieval, as expected for an effective inverted index. OpenSearch UltraWarm's FST-based secondary index outperforms LogCloud on exact matches (2.5x geomean faster on average), except for Windows. However, LogCloud is 3.6x faster on substring queries which bypass OpenSearch's FST index, achieving up to 7.5x speedup on the largest Cluster dataset. Most notably, LogCloud matches this performance using only S3 storage instead of OpenSearch's disk/RAM-based indices, demonstrating competitive serverless query performance without the costs of warm storage.

On the small dataset Hdfs, LogGrep performs better than LogCloud since the compressed logs can be quickly downloaded and scanned. However, it performs over 10x worse on the larger datasets like Hadoop and Cluster, where the search time on disk actually eclipses download time. Since LogGrep does not rely on indices, it has to exhaustively scan all the variables, causing its poor scalability. As a result, on URI queries, LogCloud is 22x geomean faster than LogGrep for Hadoop and 12x for Cluster.

5.3 Total Cost of Ownership

Our analysis so far across log volumes ranging from 1.5GB to 429GB demonstrates that LogCloud consistently maintains a middle ground between LogGrep and OpenSearch in terms of storage footprint and search latency. We now examine how these performance characteristics, combined with indexing costs, influence the total cost of ownership (TCO) of the log management system.

The precise question we seek to answer is given a log dataset and a fixed operating horizon, say 12 months, what is the most cost effective system for a particular total query load? To answer this question, we estimate the TCO of OpenSearch, LogGrep and LogCloud as follows:

- **OpenSearch UltraWarm** is typically operated as a longrunning cluster, where the cost consists of many components such as searcher nodes, UltraWarm nodes, EBS cost and S3 cost for UltraWarm. While the operating cost of the smallest OpenSearch

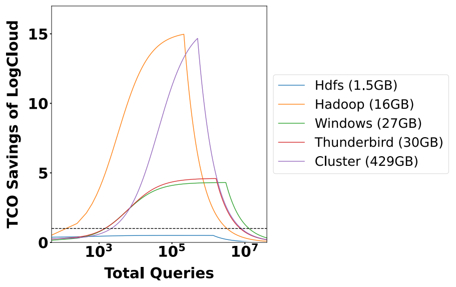


Figure 8: LogCloud's TCO savings compared to the cheaper of LogGrep and OpenSearch UltraWarm at 12 months. Each solid curve corresponds to a different log type. The line at 1 represents where LogCloud outperforms baselines.

cluster could easily exceed \$1500/month [5], we adopt a strict lower bound of this TCO here, which is just the storage cost of the primary shard of the index in S3 and the smallest required UltraWarm node (\$174/month), ignoring ingestion cost.

- **LogGrep**'s cost can be estimated as: compression cost (compression time × cost of EC2 instance) + storage cost (compressed logs size × S3 cost) + search cost (*representative search latency* × cost of EC2 searcher instance × the total number of queries).
- **LogCloud**'s operating cost can be computed with the same components as LogGrep. In this case, the storage footprint contains both the Parquet and the LogCloud index. LogCloud builds more indices on top of LogGrep, having a 10.7x geomean higher indexing cost on all the log types. However, the absolute cost is still quite low, only \$3.6 on the 429GB Cluster dataset and less than \$1 on all other datasets.
- In addition, we add another TCO comparison against **Datadog Flex logs**. We use the Flex "Starter" pricing at \$0.1/GB ingested and \$0.6 per million records per month. Datadog also offers Flex pricing with \$0.05 per million record per month with a fixed compute commitment, though the smallest such commitment would exceed the cost of all other systems considered here [49]. We also considered Grafana Loki [26], though we found LogGrep to be more economical in all cases.

We estimate the representative search latency for LogGrep and LogCloud here by averaging the latencies of different types of queries in our benchmark. Based on this estimate of the representative search latency, we can then compute the TCO at different query loads. In Figure 8, we plot LogCloud's TCO saving over the next best approach at different query loads for the different log types with a 12 months operating horizon. We make two observations:

First, the plot exhibits a distinctive peak shape. OpenSearch UltraWarm and Datadog have free search cost but high storage cost, LogGrep is the opposite, while LogCloud is in between the two approaches. As a result, at very low query loads LogGrep is more cost efficient whereas at high query loads OpenSearch is more efficient. The peak occurs where OpenSearch or Datadog surpasses LogGrep in terms of cost efficiency: we find OpenSearch to be more cost efficient than Datadog for Thunderbird and Cluster, with Datadog better on the other three datasets. These systems surpass LogCloud in cost efficiency at around 10^7 queries.

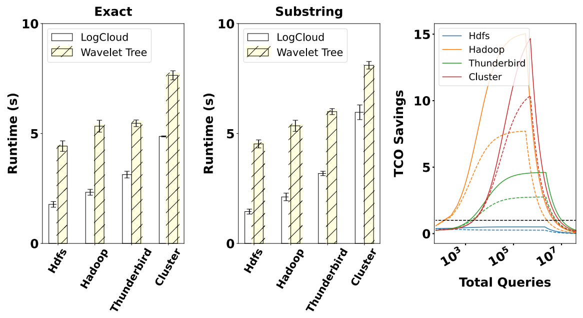


Figure 9: LogCloud index search times for exact and substring queries with custom FM-index vs wavelet tree. The solid line denotes the TCO profile of the custom FM-index, whereas the dashed line indicates that of the baseline wavelet tree.

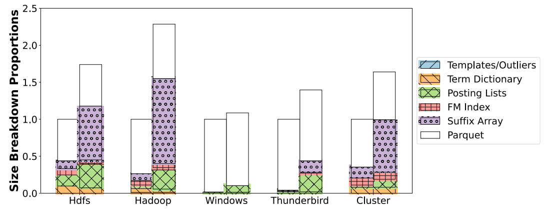


Figure 10: LogCloud component sizes with (left bars) and without (right bars) the range reduction optimization.

Second, for large-scale datasets like Hadoop and Cluster, LogCloud achieves optimal cost-efficiency in a "sweet spot" query volume range spanning around four magnitude from 1000 to 10^7 total queries, where it can be up to 15x cost-effective than alternatives. Importantly, we note that the total range of queries where LogCloud wins is consistent across the different log types, enabling practitioners to reliably predict its effectiveness for new log sources.

5.4 Ablation Studies

LogCloud's main technical novelty lies in the custom FM-index described in Section 3. In Figure 9, we show LogCloud's index search time with an optimized wavelet tree implementation [23, 28]. The subsequent Parquet access speed is not compared as it is the same between the two strategies, which download the same pages. We skip this analysis for the Windows dataset as no secondary index was constructed. Across the remaining queries, our custom FM-index achieves a geomean 2.2x speedup over the wavelet tree baseline, significantly increasing LogCloud's TCO advantage.

In Section 3.2, we introduce the range reduction optimization that compresses the suffix array by storing term dictionary chunk numbers instead of individual term offsets. In Figure 10, we show that for three log types, the suffix array was the largest component before this optimization. The optimization reduces the suffix arrays' size by geomean 8.8x, after which the Parquet files dominate the storage footprint, marking diminishing returns for further index size optimizations. Tuning the term dictionary chunk size provides direct control over the suffix array size - Figure 11a shows we can reduce it by nearly 90% (from 286MB to 30MB) simply by adjusting

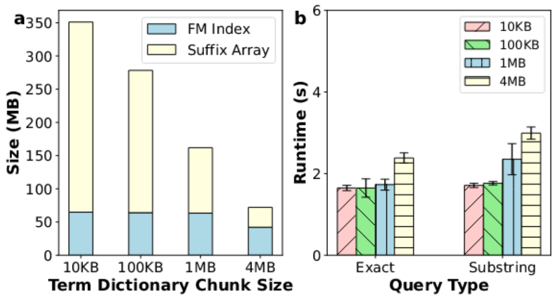


Figure 11: a) Term dictionary target chunk size vs index size. b) Search latency by type vs. term dictionary chunk size.

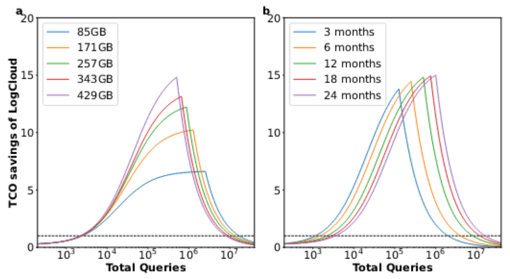


Figure 12: How the TCO ratio curve for Cluster shifts at a) different dataset scales and b) different operating horizons.

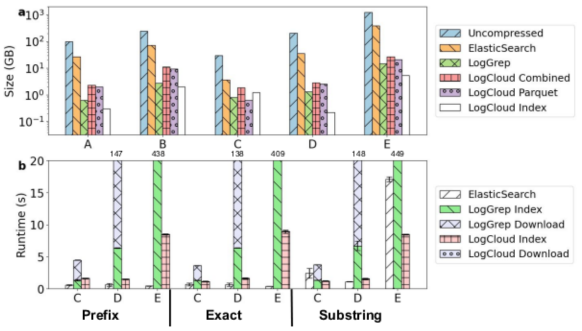


Figure 13: a) Storage footprint and b) Search times of ElasticSearch, LogGrep and LogCloud.

the chunk size from 10KB to 4MB, while the FM-index size remains relatively stable. However, this reduction has a tradeoff: as shown in Figure 11b, larger chunks require more exhaustive scanning during searches, increasing substring query times from 1.7s to 3s.

5.5 Scalability

Our evaluation shows that LogCloud maintains both low storage footprint and search latency scaling to datasets up to 429GB, achieving up to 10x TCO savings by avoiding OpenSearch and LogGrep's poor scalability at high scale due to the lack of appropriate indexing support. In this section, we control for the variability in log content and examine LogCloud's scaling along two axes, dataset size and operating horizon, by examining just the Cluster dataset.

In Figure 12a, we plot the TCO savings curve for the Cluster dataset at different subsampled sizes. We see that the cost benefits of LogCloud increases significantly at larger dataset sizes over a stable range of total queries of around four orders of magnitude, confirming LogCloud's advantage at higher scales over OpenSearch UltraWarm and brute-force scanning with LogGrep.

Figure 12b shows how operating horizons affect LogCloud's TCO curve. LogCloud's cost advantages increase with longer durations before converging. With longer operating times, LogCloud becomes cost-effective at higher query volumes but maintains advantages at high loads. This shift is beneficial since longer operations typically involve higher query volumes.

5.6 Production Test Case

We also tested LogCloud on the real production logs generated by a hosted service of a major public cloud provider, with the biggest around 1.2TB in size. The logs are produced in Json format and currently stored and queried in self-hosted ElasticSearch hot-tier with a set retention period. We tested LogGrep and LogCloud using the same configurations as the open source datasets.

Figure 13 shows that similar to the public log datasets, LogCloud significantly reduces storage footprint compared to ElasticSearch, achieving geomean 8x on the five datasets. We also show the performance of nine URI substring queries on log types C, D and E. As expected, LogGrep performs acceptably on the smaller dataset C, but is much worse than the other two options on the larger datasets D and E. In contrast to OpenSearch UltraWarm used on the public log datasets, the ElasticSearch hot-tier service stores the index entirely in memory/SSD, leading to even lower search latencies for prefix and exact URI queries. However, it is still significantly slower for substring queries, particularly for very large datasets such as E. For these queries, we see LogCloud's pure object-storage based design can still outperform ElasticSearch by geomean 2x.

6 RELATED WORK AND CONCLUSION

In conclusion, LogCloud represents an emerging third category of cloud OLAP systems: indexed disaggregated systems, which strikes a middle ground between compute-storage integrated systems like ElasticSearch, OpenSearch and Splunk suitable for very high query loads [20, 39, 48] and compute-storage disaggregated systems like BigQuery, LogGrep, and data lakehouses which excel at low query loads [1, 4, 8, 14, 52]. Other indexed disaggregated systems include Apache Hudi's B-Tree indices and Hyperspace's external indices [41, 42, 54]. While they primarily focus on developing scalable algorithms for key lookup using tree-based indices or materialized views to speed up SQL operations, LogCloud explores specialized indices optimized for log search operations. We have open-sourced LogCloud[1] to enable further research.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant 62025203 and in part by the National Science Foundation under Grant CCF-2216964. This work used AWS through the CloudBank project, which is supported by National Science Foundation grant #1925001.

[1]https://github.com/marsupialtail/rottnest/

REFERENCES

[1] 2024. Apache Iceberg. https://iceberg.apache.org/. Accessed: 2024-01-23.

[2] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. 2015. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 337–350.

[3] Amazon Web Services. 2023. Optimizing S3 Performance. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance.html

[4] Amazon Web Services. 2024. Amazon Athena. https://aws.amazon.com/athena/ Accessed: 2024-06-29.

[5] Amazon Web Services. 2024. *Amazon OpenSearch Service Pricing*. Amazon Web Services. https://aws.amazon.com/opensearch-service/pricing/ Accessed: 2024-12-09.

[6] Amazon Web Services. 2024. Orca Security's Journey to a Petabyte-Scale Data Lake with Apache Iceberg and AWS Analytics. AWS Big Data Blog. https://aws.amazon.com/blogs/big-data/orca-securitys-journey-to-a-petabyte-scale-data-lake-with-apache-iceberg-and-aws-analytics/ Accessed: December 2024.

[7] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[8] Michael Armbrust, Ali Ghodsi, Reynold Xin, and Matei Zaharia. 2021. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of CIDR*, Vol. 8. 28.

[9] Mark Atkins. 2021. *Find strings within strings faster with the new Elasticsearch wildcard field*. Elastic Blog. https://www.elastic.co/blog/find-strings-within-strings-faster-with-the-new-elasticsearch-wildcard-field

[10] Philip Bille, Anders Roy Christiansen, Patrick Hagge Cording, and Inge Li Gørtz. 2015. Finger search in grammar-compressed strings. *arXiv preprint arXiv:1507.02853* (2015).

[11] Michael Burrows. 1994. A block-sorting lossless data compression algorithm. *SRS Research Report* 124 (1994).

[12] Francisco Claude and Gonzalo Navarro. 2011. Self-indexed grammar-based compression. *Fundamenta Informaticae* 111, 3 (2011), 313–337.

[13] Cribl. 2024. Parquet Schemas - Cribl Stream Documentation. https://docs.cribl.io/stream/4.3/parquet-schemas/ Accessed: December 2024.

[14] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.

[15] DataDog. 2023. *DataDog*. https://www.datadoghq.com/

[16] Datadog. 2024. Datadog Flex Logs. Datadog Documentation. https://docs.datadoghq.com/logs/log_configuration/flex_logs/ Accessed: 2024-02-23.

[17] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. 2017. Optimizing Space Amplification in RocksDB.. In *CIDR*, Vol. 3. 3.

[18] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2769–2782.

[19] Elastic. 2023. *Elastic Kibana*. https://www.elastic.co/kibana

[20] Elastic. 2023. *Elasticsearch Platform — Find real-time answers at scale*. https://www.elastic.co/

[21] Facebook. 2023. Zstandard - Fast real-time compression algorithm. https://github.com/facebook/zstd. Original-source code available at https://github.com/facebook/zstd.

[22] Paolo Ferragina and Giovanni Manzini. 2000. Opportunistic data structures with applications. In *Proceedings 41st annual symposium on foundations of computer science*. IEEE, 390–398.

[23] Paolo Ferragina and Giovanni Manzini. 2005. Indexing compressed text. *J. ACM* 52, 4 (2005), 552–581.

[24] Simon Gog, Juha Kärkkäinen, Dominik Kempa, Matthias Petri, and Simon J Puglisi. 2019. Fixed block compression boosting in FM-indexes: Theory and practice. *Algorithmica* 81 (2019), 1370–1391.

[25] Grafana. 2023. *Grafana Loki OSS | Log aggregation system*. https://grafana.com/oss/loki/

[26] Grafana Labs. 2024. Understanding Grafana Cloud Logs Billing. Grafana Cloud Documentation. https://grafana.com/docs/grafana-cloud/cost-management-and-billing/understand-your-invoice/logs-invoice/ Accessed: 2024-02-23.

[27] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. 2003. High-order entropy-compressed text indexes. (2003).

[28] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. 2011. Wavelet trees: From theory to practice. In *2011 First International Conference on Data Compression, Communications and Processing*. IEEE, 210–221.

[29] Guy Joseph Jacobson. 1988. *Succinct static data structures*. Carnegie Mellon University.

[30] Suman Karumuri, Franco Solleza, Stan Zdonik, and Nesime Tatbul. 2021. Towards observability data management at scale. *ACM SIGMOD Record* 49, 4 (2021), 18–23.

[31] Julian Labeit, Julian Shun, and Guy E Blelloch. 2017. Parallel lightweight wavelet tree, suffix array and FM-index construction. *Journal of Discrete Algorithms* 43 (2017), 2–17.

[32] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review* 44, 2 (2010), 35–40.

[33] Heng Li and Richard Durbin. 2009. Fast and accurate short read alignment with Burrows–Wheeler transform. *bioinformatics* 25, 14 (2009), 1754–1760.

[34] M3DB. 2023. *M3: Open Source Metrics Engine*. https://m3db.io/

[35] Christos Makris. 2012. Wavelet trees: A survey. *Computer Science and Information Systems* 9, 2 (2012), 585–625.

[36] Mehryar Mohri. 2004. Weighted finite-state transducer algorithms. an overview. *Formal Languages and Applications* (2004), 551–563.

[37] Yuta Mori. [n. d.]. libdivsufsort: A lightweight suffix sorting library. https://github.com/y-256/libdivsufsort. Accessed: 2024-06-22.

[38] Craig G Nevill-Manning and Ian H Witten. 1997. Identifying hierarchical structure in sequences: A linear-time algorithm. *Journal of Artificial Intelligence Research* 7 (1997), 67–82.

[39] OpenSearch. 2023. *OpenSearch*. https://www.opensearch.org/

[40] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.

[41] Rahul Potharaju, Terry Kim, Eunjin Song, Wentao Wu, Lev Novik, Apoorve Dave, Andrew Fogarty, Pouria Pirzadeh, Vidip Acharya, Gurleen Dhody, et al. 2021. Hyperspace: The indexing subsystem of azure synapse. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3043–3055.

[42] Rahul Potharaju, Terry Kim, Wentao Wu, Vidip Acharya, Steve Suh, Andrew Fogarty, Apoorve Dave, Sinduja Ramanujam, Tomas Talius, Lev Novik, et al. 2020. Helios: hyperscale indexing for the cloud & edge. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3231–3244.

[43] Quickwit. 2023. *Quickwit*. https://quickwit.io/

[44] 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)*. 183–198.

[45] Amazon Web Services. 2021. *Security Data Management - Amazon Security Lake*. https://aws.amazon.com/security-lake/

[46] Splunk. 2022. *How to extract bunch of UUIDs from a string using regex*. Splunk Community. https://community.splunk.com/t5/Splunk-Search/How-to-extract-bunch-of-UUIDs-from-a-string-using-regex/m-p/622971

[47] Splunk. 2023. *Splunk*. https://www.splunk.com/

[48] Splunk Inc. 2024. Indexing and search architecture. https://lantern.splunk.com/Splunk_Success_Framework/Platform_Management/Indexing_and_search_architecture Accessed: 2024-07-03.

[49] Sumo Logic. 2023. What You Should Know About Datadog Flex Logs and Pricing. Sumo Logic Blog. https://www.sumologic.com/blog/should-know-about-datadog-flex-logs/ Accessed: 2024-02-23.

[50] Joris van der Walker. 2023. The Burrows-Wheeler Transform. https://curiouscoding.nl/posts/bwt/. https://curiouscoding.nl/posts/bwt/ Accessed on 2024-10-20.

[51] Vantage. 2024. AWS EC2 r6i.xlarge On-Demand Instance Pricing. https://instances.vantage.sh/aws/ec2/r6i.xlarge. https://instances.vantage.sh/aws/ec2/r6i.xlarge Accessed on [Insert Access Date].

[52] 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*. 452–468.

[53] Junyu Wei, Guangyan Zhang, Yang Wang, Zhiwei Liu, Zhanyang Zhu, Junchao Chen, Tingtao Sun, and Qi Zhou. 2021. On the Feasibility of Parser-based Log Compression in {Large-Scale} Cloud Systems. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 249–262.

[54] Shiyan Xu and Sivabalan Narayanan. 2023. Record Level Index: Hudi's blazing fast indexing for large-scale datasets. https://hudi.apache.org/blog/2023/11/01/record-level-index/. Accessed: 2024-07-05.

[55] Feng Zhang, Jidong Zhai, Xipeng Shen, Onur Mutlu, and Wenguang Chen. 2018. Efficient document analytics on compressed data: Method, challenges, algorithms, insights. *Proceedings of the VLDB Endowment* 11, 11 (2018), 1522–1535.

[56] Feng Zhang, Jidong Zhai, Xipeng Shen, Dalin Wang, Zheng Chen, Onur Mutlu, Wenguang Chen, and Xiaoyong Du. 2021. TADOC: Text analytics directly on compression. *The VLDB Journal* 30 (2021), 163–188.

[57] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R Lyu. 2020. Loghub: A Large Collection of System Log Datasets for AI-driven Log Analytics. *arXiv e-prints* (2020), arXiv–2008.