LogCloud: Interactive Log Retrieval on Object Storage with the FM-Index (Flavor: Systems)

Anonymous Author(s)

ABSTRACT

Large organizations can easily emit TBs of logs every day. Users often need to conduct interactive searches on these logs to troubleshoot issues or detect threats. Organizations would also like to control the cost to store and query these logs, which could exceed tens of millions of dollars per year. Current systems either perform full-text indexing like ElasticSearch to provide efficient keyword search at the expense of high storage cost, or store unindexed compressed logs like Loki and LogGrep to provide low storage cost with slower searches.

In this paper, we propose LogCloud, a new object-storage based log management system that supports both cheap compressed log storage and efficient indexed search. LogCloud is based on LogGrep, but constructs inverted indices on the extracted variable tokens. Instead of using FSTs or SSTables for the secondary index on the term dictionary, we employ the FM-index based on the Burrows-Wheeler Transform to support efficient substring matching. Experiments on five public log datasets show that LogCloud can achieve similar keyword search performance compared to OpenSearch UltraWarm with an order of magnitude lower storage cost.

ACM Reference Format:
Anonymous Author(s). 2024. LogCloud: Interactive Log Retrieval on Object Storage with the FM-Index (Flavor: Systems). In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern organizations generate TBs of logs every day. These logs can be divided into three categories as in [31]: online logs in the last few minutes or hours that send live alerts, near-line logs in the last days to months that are queried on demand when a problem occurs or is suspected, and offline logs after a few months which are typically archived for compliance reasons.

Of these three type of logs, we believe near-line log management to be the most challenging. Online logs are frequently queried but the size is typically small, allowing straightforward solutions such as real-time datastores such as ElasticSearch hot indices [7]. Offline logs can be PBs in size but rarely queried, a great fit for archival services like AWS Glacier after high compression by log specific methods [20, 32]. In comparison, near-line logs can also easily be multiple TBs in size, but also must be interactively queried through a dashboard such as Kibana, Datadog or Splunk for troubleshooting or cybersecurity use cases [6, 8, 29]. Organizations would like to do both of the following:

- **Cheap Search** It must be possible to cheaply and interactively search these logs with full text search queries that might involve prefix (req-12345*), suffix (*.amazon.com), or substring searches (*.amazon.*).
- **Cheap Storage** The logs should be stored on object storage in a highly compressed format, with minimal disk/RAM usage. Disk is much more expensive than object storage on the cloud. For example, it costs \$240 to store 1 TB in EBS compared to \$23 on S3 for 30 days.[1]

Current systems struggle to satisfy both objectives. Popular solutions like Splunk, ElasticSearch and OpenSearch provide fast query times by constructing a full text search index on the logs. However, these indexes tend to demand a very high storage footprint due to large index size, making storage expensive [23, 25, 29, 31].

Another approach popular in industry is storing unindexed compressed logs in systems like Loki [12]. Recent works such as CLP and LogGrep have shown logs can be highly compressed by exploiting static and runtime patterns [25, 31]. While these solutions can often achieve an order of magnitude less storage footprint than ElasticSearch by forgoing the full text indices, search becomes expensive since a large volume of text now needs to be decompressed and scanned.

For example, LogGrep breaks down logs into *templates* and *variable tokens*, such as UUIDs. During search, variable tokens matching type- and length-based filtering are scanned exhaustively. A search on a Kubernetes pod ID still scans all the (potentially billions) Kubernetes pod IDs in the entire dataset, since they are all of the same type and length. Assuming the compressed logs are stored on object storage, such exhaustive scans require high network bandwidth and lead to high search costs.

We propose LogCloud, a system that addresses both requirements. LogCloud combines ideas from ElasticSearch and LogGrep. It first uses LogGrep to break down logs into templates and variable tokens. It then constructs full text indices on the variable tokens only. The main technical challenge addressed in LogCloud is coming up with an inverted index design that supports efficient substring search on object storage. Common data structures like finite state transducers and sorted string tables only efficiently support prefix searches. To address this limitation, we turn to the FM-index based on the Burrows Wheeler Transform (BWT), but it has mostly been applied only in a disk/RAM setting [1, 5, 13]. LogCloud proposes a novel object-storage based FM-index implementation with corresponding inverted index optimizations that allows fast substring searches with small storage footprint.

Conference'17, July 2017, Washington, DC, USA
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…\$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]Assuming three-way replication on EBS since it has multiple orders of magnitude higher failure rates than S3.

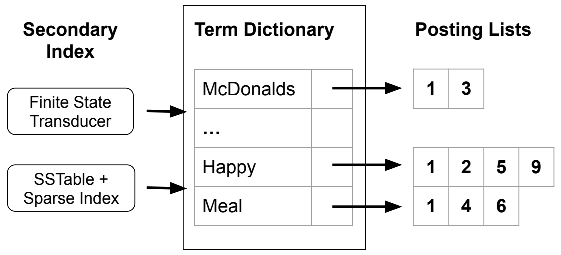

Figure 1: Typical design of an inverted index used in search engines like ElasticSearch.

We evaluate LogCloud on a benchmark suite of five public log datasets and four query types against OpenSearch and LogGrep. We show that LogCloud achieves better search speed than OpenSearch and same order-of-magnitude storage footprint as LogGrep, achieving the best-of-both-worlds of fast search and cheap storage.

This paper makes two key contributions:

- We propose a novel object-storage based full text index based on a custom BWT FM-index that can support efficient substring search with very low storage footprint.
- We implement a complete log management system, LogCloud, that can efficiently support text search queries on highly compressed logs stored on object storage with an order of magnitude less cost than comparable systems such as OpenSearch.

2 MOTIVATIONS FROM RELATED WORK

Full text indexing solutions like ElasticSearch and OpenSearch have a high storage footprint, while unindexed solutions like Loki and LogGrep shift the cost to query time. The key question we aim to address in this work is: **how to achieve both fast search and cheap storage on compressed logs on object storage?**

Recall that LogGrep has high search cost because it has to exhaustively search through the extracted variable tokens. We believe the natural solution is to ***build a full text index on those extracted tokens***. This index should ideally be smaller than the compressed logs themselves, and can be efficiently queried directly on object storage through random accesses so it does not have to be downloaded entirely for every search.

LogGrep divides variable tokens based on what characters they contain. A variable token could have one of 63 types, denoting if it contains numbers, lower or upper case characters, symbols, or any combinations thereof [31]. Variable tokens are grouped into types, and during search time only variable tokens of compatible types are scanned. For example, if the query contains letters, the type group that only contain numbers can be skipped. In LogCloud, we build a full text index on each type group.

2.1 Full Text Indices

How should we build this full text index? The most popular approach today is an *inverted index* shown in Figure 1, which maps each token in the text corpus to a *posting list* of locations where the token appears in the corpus. Most inverted indices also have an associated secondary index to efficiently look up a token in the term dictionary, which could be GBs or even TBs in size. This is especially true in log search scenarios, where these tokens could correspond to UUIDs like Kubernetes pod IDs and request IDs.

There are two popular ways to build this secondary index, which effectively maps string keys to values corresponding to the location of the posting lists. Finite state transducers (FST) are used by systems like ElasticSearch, OpenSearch and M3DB [7, 21, 23]. An FST maps a set of key value pairs to a directed acyclic graph. Querying the value corresponding to a key can be done through traversing the graph [4]. Another popular approach is based on sorted string tables (SSTables). This approach is adopted by KV stores like Cassandra and newer search engines like Quickwit [17, 24]. In this approach, sorted chunks of keys are stored alongside their corresponding values. A sparse index is built on the sorted chunks to quickly locate which chunk contains the query key. The chunk is then downloaded and searched exhaustively.

Both FSTs and SSTables allow users to perform key searches on potentially billions of keys with just a few random reads from the data structure. In the case of FSTs, the number of reads is O(length of the query) while in the case of SSTables it is the log of the number of chunks, assuming a B-tree is used for the sparse index. In addition to efficient exact match queries, both FSTs and SSTables can easily support prefix queries and suffix queries with adaptations[2].

However, they have a critical limitation, which is the lack of support for substring queries. Some aforementioned systems like Quickwit or Cassandra do not support this functionality outright, while others like ElasticSearch cannot efficiently use the secondary index and must perform expensive additional scans of the term dictionary [7, 17, 24].

Substring searches can be critical for observability and cybersecurity use cases, e.g. looking for a partial IP address match. In other cases, these searches are the default as the user, an SRE engineer or security analyst, is typically not expected to be familiar with the tokenization scheme used by the search engine. As a result, what the user thought might be a prefix query might actually require a wildcard query. Industrial systems like Splunk and Datadog support these queries out of the box [6, 29].

Another important reason why substring queries are important is that it allows us to use a more limited set of delimiters in LogGrep when parsing out variable tokens. In log management scenarios, variables such as IP addresses and Kubernetes pod names often include delimiter tokens like periods or dashes. Consider Kubernetes pod names such as "nginx-554b9c67f9-c5cv4". If we treat the dash as a delimiter, common replicaset queries like '554b9c67f9' become exact match queries, reducing the need for substring queries. However, this approach turns exact-match queries like 'nginx-554b9c67f9-c5cv4' into multi-term queries, divided into three tokens, significantly increasing complexity and cost. It is much simpler to treat the entire pod name "nginx-554b9c67f9-c5cv4" as one token, and perform substring queries to retrieve "554b9c67f9" or "nginx-554b9c67f9-c5cv4".

[2]The FST or SSTables can simply be built from reversed keys.

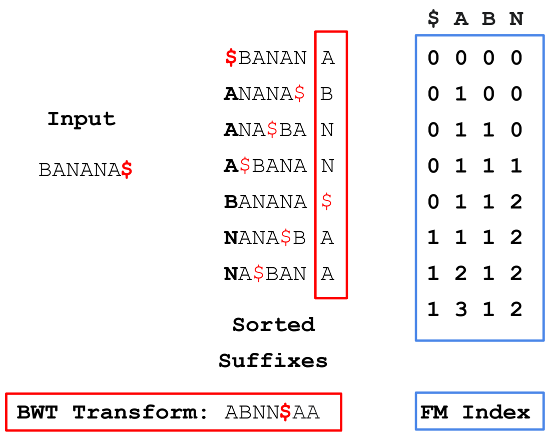

Figure 2: Summary of BWT and FM-index on the input string banana. All suffixes are sorted and the BWT is the marked in the red box. The FM-index denotes the count of each character up to any position in the BWT.

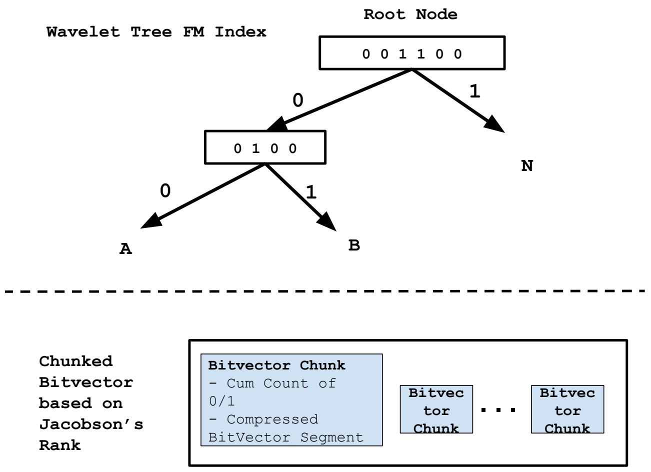

Figure 3: Wavelet tree index with bitvector storage based on Jacobsen's rank. Each leaf node corresponds to a character in the alphabet, where the path to the leaf node corresponds to the character's binary encoding. For example, to compute $rank(B, 4)$, we first lookup B's binary representation 01. 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.

2.2 The BWT and FM-index

The question now boils down to: **what object-storage based secondary index to build to allow efficient substring searches on the term dictionary?** To efficiently handle substring searches, LogCloud uses the Burrows-Wheeler Transform (BWT). The BWT is a data transformation used primarily in data compression and string matching algorithms, especially in bioformatics where short reads need to be aligned to a large immutable reference genome [5, 19]. Recent work has also explored its potential in other applications like key-value stores [1].

An example of the BWT is shown in Figure 2. To obtain the BWT of an input text corpus[3], first generate a matrix of cyclic permutations of the corpus. Then, sort these permutations lexicographically based on the suffix and extract the last column from the array of suffixes. This last column of characters, of the same length as the input text corpus, is defined as the BWT of the input. The BWT often exhibits runs of similar characters, which is beneficial for compression algorithms. More importantly for our application, the BWT is also a fundamental step in the construction of the FM-index, which allows for efficient substring searches [11].

The FM-index is an abstract data structure that allows for looking up the rank of a character in the BWT, i.e. $rank(c, i)$ is defined as how many times character c has appeared up to position i. This character rank operation can be used as a key primitive in substring matching in the original text corpus. It is well-known Algorithm 1 can be used to find all occurrences of a substring P in the input text corpus using the rank operation repeatedly.

The core algorithmic decision in any BWT application is how to implement this abstract FM-index. The most popular implementation targeting RAM or disk based scenarios is a *wavelet tree* [13, 16, 22]. The wavelet tree compresses the BWT into a binary tree, where each node contains a bitvector. An example wavelet tree

[3]which could consist of billions of characters, like the human genome

Algorithm 1 Iterative Substring Search using FM-index with BWT

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

for the BWT shown in Figure 2 is illustrated in Figure 3. To retrieve the rank of a character, the tree is traversed from the root with rank operations done on the bitvectors at each node, as described in Algorithm 2.

As illustrated in Figure 2, the bitvector at each node is typically stored in chunks so an algorithm like Jacobson's Rank can be used to efficiently compute the bitvector rank. Each chunk contains a fixed size portion of the bitvector together with the ranks of 0 and 1 up to the start of the chunk. To compute $rank(0, i)$ on bitvector with chunk size C, chunk i/C is retrieved: $rank(0, i) =$

Algorithm 2 Compute rank using a Wavelet Tree

```
1: function Rank($c$, $i$, WaveletTree)
2:     $node \leftarrow$ WaveletTree.root
3:     $bits \leftarrow$ binary encoding of $c$, length typically corresponds to entropy of alphabet
4:     for $bit$ in $bits$ do
5:         $i \leftarrow node.rank(bit, i)$
6:         $node \leftarrow bit == 0\ ?\ node.left : node.right$
7:     end for
8:     return $i$
9: end function
```

$x + rank(0, i\%C)$ on chunk i/C, where x is the rank of 0 up to chunk i/C stored in chunk i/C.

The result of Algorithm 1 indicates the query pattern is found between 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 simple list that records the offset in the original corpus that corresponds to each position in the BWT. However, this would be 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 offset array, which stores only offsets for every K positions. If a position i's mapping is not stored, the FM-index has to be repeatedly consulted to relate position i's mapping to $i - 1$'s mapping, until a sampled location is hit.

2.3 Speed and Size Challenges

To the best of our knowledge, all existing implementations of the FM-index have targeted disk or in-memory scenarios. This is because the BWT is typically used to map short reads against a reference genome, which rarely exceeds several GBs in size. However, in our scenario, the index structure resides in object storage, which has an order of magnitude higher read latency than even slow HDDs.[4] This raises two critical challenges for the standard wavelet tree FM-index implementation.

The first challenge is the cost of substring search. From Algorithm 2 we note 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 extracted variable tokens like UUIDs, which are typically random by design, 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.

From Algorithm 1, we see that we have to 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. Since each read request can take tens of milliseconds, we will be paying multiple seconds just to index into the term dictionary.

The second challenge relates to the sampled offset array, typically 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 is acceptable when the FM-index is in memory or on disk, it is unacceptable when each access incurs $O(|P|)$ random accesses as

[4]The cold read latency is tens of ms from EC2 instance to S3 in the same region.

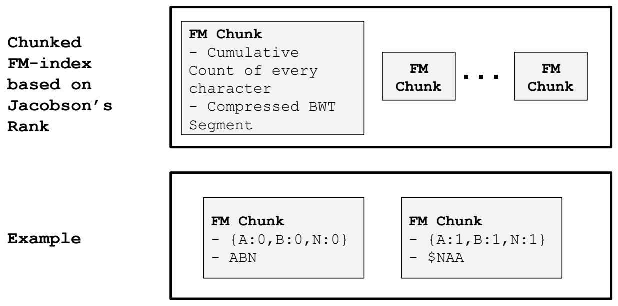

Figure 4: Chunked FM-index inspired by Jacobson's Rank. An example for the BWT in Figure 2 is shown.

discussed. This is particularly problematic as thousands of positions potentially have to be mapped. Even though mapping each position can be parallelized, making thousands of small concurrent requests to object storage is also an inefficent anti-pattern and may run into S3 request throttling [2].

3 OBJECT STORE NATIVE INVERTED INDEX

LogCloud effectively tackles the two challenges by focusing on the unique aspects of our primary task: constructing inverted indices for variable tokens on object storage. Specifically, we heavily lean on the two following properties of object storage:

- Data retrieval is significantly more expensive compared to processing the data
- Small requests are all latency bound – retrieving 1 byte and 1 MB have similar latencies.

These observations drive our development of a custom FM-index implementation and a novel solution for the BWT mapping issue. In addition, we exploit the nature of the data we are indexing, *random alphanumeric strings with heavy temporal locality*, to introduce further optimizations which speed up search through early stopping and save storage with custom posting list compression.

3.1 Fast Search with Custom FM-Index

Our first challenge is the $O(H_C|P|)$ sequential requests we have to make to search the FM-index using the wavelet tree implementation. We reduce this cost to $O(|P|)$ through using a different FM-index implementation, show in Figure 4.

Instead of constructing a wavelet tree, we divide the BWT itself into fixed-size chunks. We compress each chunk and also store the rank of every character in the BWT up to the beginning of the chunk. Thus, to compute $rank(c, i)$ with chunk size C, chunk i/C is retrieved, $rank(c, i) = x + rank(c, i\%C)$ on chunk i/C, where x is the rank of c up to chunk i/C. This algorithm is very similar to the bitvector rank algorithm based on Jacobson's rank discussed in Section 2.2, where we replace bits with characters.

This approach requires just reading one chunk to compute the rank, and is much simpler than the wavelet tree design. This implementation 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 object-storage based scenario where read cost dominates, this is the better choice.

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, on the random variable tokens we are indexing, such entropy-based encoding typically do not help. In addition, we can compress each character chunk in our FM-index using a generic compression scheme like Zstd [10], as illustrated in Figure 4, and simply decompress the chunk upon reading. Additional compression adds too much overhead for disk/RAM-based indices since read is fast, but acceptable in our case since decompressing a chunk in memory is much faster than downloading the chunk from object storage.

3.1.1 *Early exit.*

Our FM-index implementation brings down the number of random reads from $O(H_C|P|)$ to $O(|P|)$. However, we can improve this further by noticing that the BWT substring search algorithm in Algorithm 1 iteratively narrows down the search results: after the n^{th} iteration, the algorithm finds the range of positions in the BWT that matches the last n characters of the suffix in the query. We can employ early stopping when we notice that this range fails to decrease after a few successive iterations. In needle-in-the-haystack queries for a long UUID with random alphanumeric characters like ‘nginx-554b9c67f9-c5cv4‘, typically only a short suffix like ‘f9-c5cv4‘ is needed to retrieve all matching results.

3.2 Reducing Mapping Size by Range Reduction

The second challenge revolves around mapping the range in the BWT returned by Algorithm 1 back to positions in the original text corpus. In our setting, the input text corpus to the BWT is all the variable tokens of a particular type extracted by LogGrep.

The naive approach stores an integer for every position in the BWT, which is the same length as the input corpus. In general, the mapping is also poorly compressible, especially if the input text corpus consists of random UUID-like strings. We find the storage footprint of this approach unacceptable on most common logs, with the size requirement approaching the uncompressed log size. However, the commonly used sampled offset array approach which stores only the mapping every K locations require too many random reads, especially if a large range of positions is returned.

In LogCloud, we use a variation of the naive approach, except we do not store the offset into the original term dictionary. Instead, we break the term dictionary into chunks, and only record the chunk number in the mapping. Even though we still have to store the same number of integers as the naive approach, the range of each integer has been reduced by several orders of magnitude. Subsequent positions in the mapping are also now more likely to be identical, further improving compression. We call this optimization technique **range reduction**.

This optimization is motivated by the observation that byte-range GET requests on object storage up to 1MB are all latency

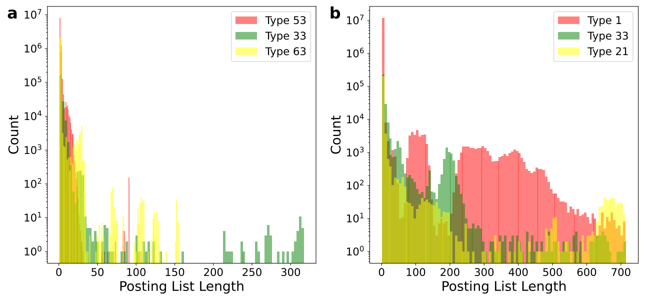

Figure 5: Histogram of posting list lengths for a) Hadoop logs dataset b) Thunderbird logs dataset. Note y-axis is logscale.

bound and have roughly the same speed[5]. 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[6] 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.

3.3 Custom Posting List Compression

After we applied the mapping size optimization, we find that the size of the posting lists often becomes the largest component in the entire index. The posting list maps a variable token in the term dictionary to the location of the original log line. We apply the range reduction philosphy again, and store the original logs themselves in 1MB chunks as row groups in Parquet files. The posting lists thus store row group numbers. An entire row group is downloaded and searched upon an index hit.

Conventional search engines have gone to great lengths to optimize the storage footprint of posting lists, which are just lists of sorted integers. Systems like ElasticSearch and M3DB typically employ Roaring Bitmaps [7, 18, 21]. However, Roaring Bitmaps are optimized for long posting lists. In our case, variable tokens like UUIDs only occur a handful of times close to each other in time, meaning they land in the same Parquet row group. This means they have only one entry in their posting lists.

In Figure 5, we show the posting list lengths distributions for two example log datasets. We see a big spike on the left at 1 followed by a long tail on the right, with the height of the tail several orders of magnitude lower than the spike. From this observation, we devise a posting list compression scheme that stores:

- A bitmap indicating whether each posting list has more than one element.
- An array of the lengths of posting lists with more than one element.
- The first element for each posting list.
- Subsequent elements from posting lists with more than one element.

Zstd compression is applied to the components. This compound compression scheme dramatically improves the compression ratio

[5]On AWS S3, from EC2 instance in same region
[6]The 1MB number is a tunable parameter that can be set for different expected network environments, e.g. datacenter vs laptop.

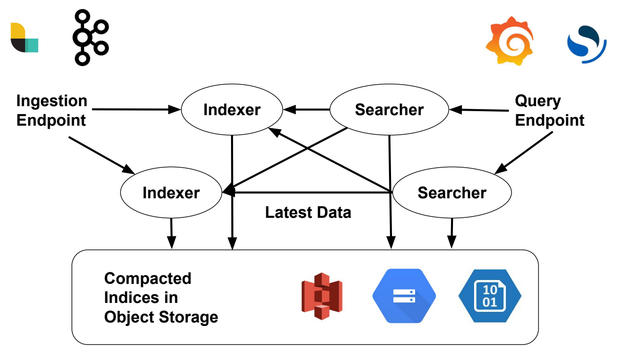

Figure 6: LogCloud Architecture. Indexers and searchers are decoupled and can be scaled separately. Most data is stored in object storage by default.

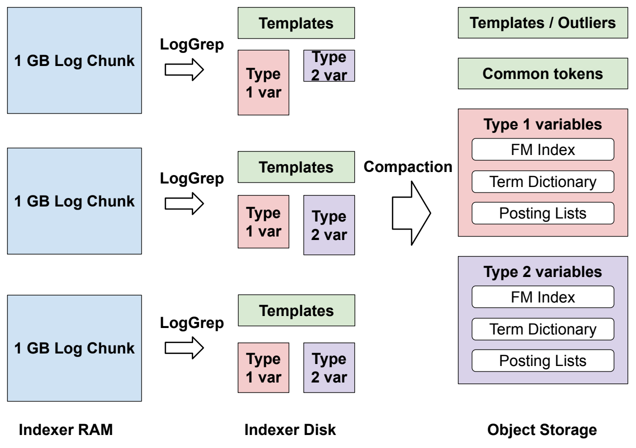

Figure 7: Indexing in LogCloud. 1GB log chunks are buffered in RAM and indexed by LogGrep. The resulting variable tokens are spilled to disk and compacted every 20 chunks. The compacted indices are uploaded to object storage along with a copy of the original logs in Parquet files.

of the posting lists compared to Roaring Bitmaps on our test datasets. After the row-group range reduction and custom compression, posting lists are now typically the smallest component in our index.

4 LOGCLOUD ARCHITECTURE

In this section, we discuss how the novel object-storage native inverted index fits in the overall architecture of LogCloud, a complete log management system. The overall architecture of LogCloud is shown in Figure 6. LogCloud decouples indexing with querying. Indexers and searchers can be scaled horizontally and independently based on ingest or query demand, which do not often line up in practice [15]. Indexers receive logs from tools like Kafka or Logstash. They keep the latest logs in memory/disk while flushing older logs and their associated indices to object storage. The query endpoint can be used with visualization tools like Grafana and OpenSearch Dashboards. Searchers directly query the compacted indices on object storage for older logs and optionally the indexers directly if the latest data must be retrieved. For our targeted use case of near-line log analytics where potentially a large historical time range needs to be probed, querying the indices is object storage is expected to dominate search performance.

4.1 Indexing

LogCloud has two key indexing steps, as shown in Figure 7. Incoming logs are stored in Parquet files with Zstd compression targeting a row group size of around 1MB. LogCloud could also index logs already in Parquet format, e.g. produced by tools like Matano or AWS Security Lake [14, 28], in which case the default row group sizes can be used.

In the first indexing step, once around 1GB of logs are collected, LogGrep is run on this group of logs to extract templates and variable parts. This is done to allow the templates to gradually change across time to reflect potential application changes to increase compression efficiency. This indexing interval can be tuned. If the interval is too small, the size of the templates and outliers become large and can dominate the actual variables; if the interval is too large, template extraction might not work as well due to slowly changing patterns in the logs across time. In our experience, 1GB works well in most cases.

The variable tokens extracted, such as Hadoop block numbers or Kubernetes pod IDs, are separated based on their LogGrep type, as previous described in Section 2. The posting lists for each variable token is then computed. As previously described in Section 3.3, the posting list is computed at Parquet row group granularity. The variable tokens and term dictionaries are stored on the disk of the indexer at this stage.

In the second indexing step, once a significant number of 1GB groups have been indexed by LogGrep, the LogGrep outputs are compacted. The templates and outliers are concatenated. Variable tokens of each type are merged along with their posting lists. Frequently occurring tokens are extracted and their posting lists are discarded. This further reduces posting list size as a long posting list that tells us to search a majority of the row group yields very little information. LogCloud simply brute force searches through all the logs if a query matches a frequently occurring token.

The secondary BWT FM-index is constructed on the compacted variable tokens of each type at this stage. The FM-index, BWT mapping, term dictionary and posting lists are then uploaded to object storage. If this compaction interval is too large, then a large volume of logs will remain on the indexers, taking up expensive disk resources. However, the smaller this interval, the more independent indexes a query has to search over a fixed time range, which directly increases search cost. In this paper, we use a value of 20GB. In principle, compacted indices could be further compacted by a background process. This is an area of future work.

Typically, the Parquet files are 10 times smaller than the raw log files, and the indexes several times smaller than the Parquets. LogCloud stores a copy of the original logs in Parquet files, which constitute the majority of its storage footprint in most cases. LogGrep does not do this, instead electing to "piece together" matching

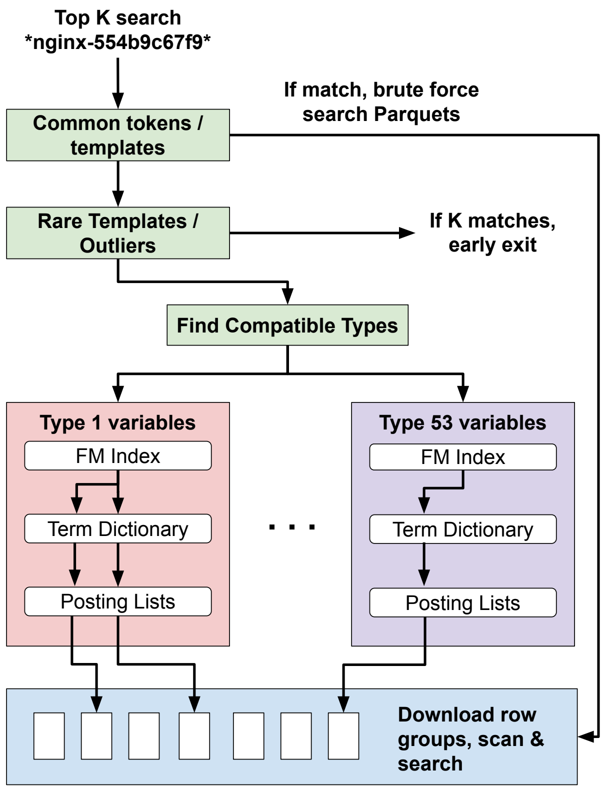

Figure 8: Searching workflow in LogCloud.

logs from their templates and variable tokens. Unfortunately we found this approach requires too many random accesses when the variable tokens are stored in object storage, making search prohibitively expensive.

In addition, storing the raw logs in Parquet format gives us easy interoperability with other systems which can ingest the Parquet files directly, like SparkSQL or AWS Athena. For example, LogCloud's index can be used with SparkSQL to speed up SQL queries that contain filters like `select count(*) from logs where message like '*abc*'`.

4.2 Searching

The key search primitive LogCloud supports is a top-K substring query on a single token, which we focus on in this section. Exact matches, prefix or suffix searches are implemented with post-processing. Multi-phrase queries are implemented by breaking it down into a conjunction of single token queries.

Searchers query the indexers directly for latest logs and the object storage directly for compacted logs. Since our paper focuses on near-line log management, we focus on the latter scenario, which is expected to dominate search costs. Searches can be trivially parallelized over different compacted indices in object storage. The search algorithm on one such index is illustrated in Figure 8.

First, common tokens and 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. If the query does not match, the searcher then downloads the rare templates and outliers from object storage. If the substring query matches enough log lines here to satisfy the top-K requirement, the search exits with these results.

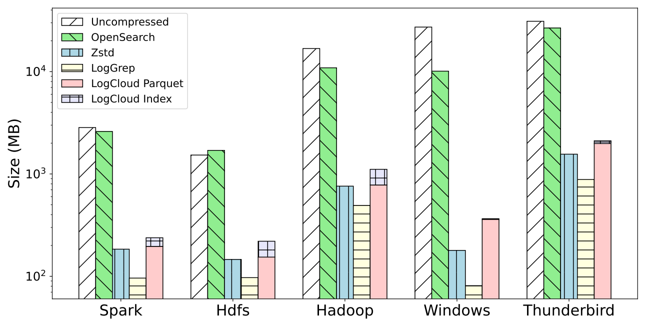

Figure 9: Storage footprint of all log management solutions considered. For LogCloud we show the size of the Parquet files vs. the index size.

If the search proceeds past this point, the searcher has to search the variable token indices. The type of the query is determined and the indices for all "compatible" types are searched.[7] For each type, the secondary BWT FM-index is searched to point to term dictionary chunks that contain the matching tokens. These chunks are downloaded and exhaustively searched. The posting lists for the matching tokens are then decompressed to retrieve the row group numbers of logs that contain these tokens in the Parquet files. These row groups are then downloaded and searched exhaustively for the original query.

LogCloud's default mode of operation puts all index data on object storage, including information on common tokens, templates and outliers. Such information is typically small and accessed for every single query, making it a good candidate for disk caching. In addition, certain metadata from the inverted indices can also be cached, like data structure offsets etc. Although LogCloud supports these forms of caching, we do not explore disk caching in this paper to simplify the evaluation against other systems.

5 RESULTS

We evaluate LogCloud against two baselines, LogGrep and AWS-managed OpenSearch service on compression and search performance. We use the LogHub dataset, similar to LogGrep [31, 35], but we filter for only datasets larger than 1GB: HDFS (1.5GB), Spark (2.4GB), Thunderbird (30GB), Hadoop (17GB) and Windows (26GB).

We test four different search queries on each of the datasets: a common keyword, an exact-match UUID query, a prefix UUID query and a substring UUID query. For example, on the Spark dataset, the queries are "ERROR", "rdd_573_3", "rdd_573*" and "*573*". The rest of the queries are in the Appendix. All are queries an engineer might reasonably be expected to search when debugging an issue

[7]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 in LogGrep) since the query could be a numeric substring of an alphanumeric token.

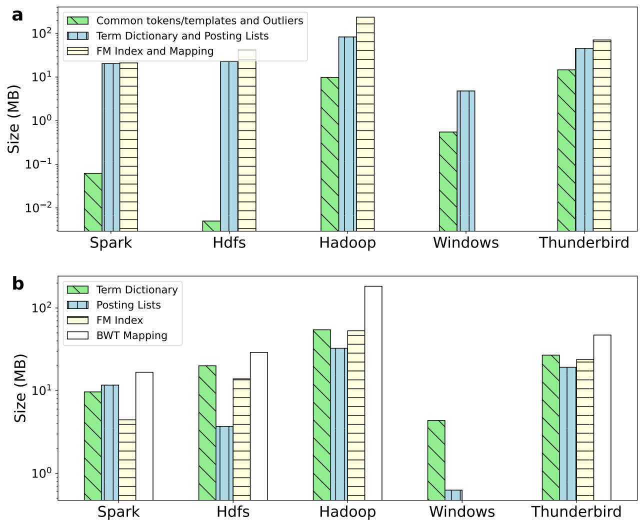

Figure 10: a) Sizes of different index components for LogCloud. b) A fine-grain breakdown of component sizes for the term dictionary, posting lists, the FM-index and the BWT mapping.

related to a particular job. For example, engineers commonly resort to wildcard queries such as "*573*" if they know certain unique identifiers must be printed in the logs, but are not familiar with the exact string format of the log files. For all three systems, we return the top 1000 results for each query.

For LogCloud we benchmark search performance on one r6i.large EC2 instance with 2 vCPUs and 16 GB of RAM. The indices and log Parquet files are stored on AWS S3 in the same region as the EC2 instance. We do not parallelize the search over multiple indices. Since searches can be easily distributed across multiple indices, this measures the "normalized search time per index". Given our design which treats each 20GB index independently, it is also not necessary to test on much larger datasets since the compression performance will stay the same and the search performance will linearly increase.

To make LogGrep work with object storage, we experimented with modifying it to query compressed logs on S3 directly. However, we were not able to get better performance than simply downloading the entire compressed output and then searching on disk, which we adopt as our baseline results here. This is mostly because LogGrep has to exhaustively scan variable tokens and some other internal design choices in LogGrep that assume efficient random reads.[8] We modify LogGrep to use Zstd compression instead of LZMA to normalize the compression algorithm used across LogGrep and LogCloud [10]. The LogGrep search experiments are conducted on the same r6i.large instance.

For AWS OpenSearch, we benchmark search speed and storage footprint from the object-storage based UltraWarm tier, which is typically used for near-line log management in industry. Since AWS OpenSearch requires using (at least three of) its own ultrawarm nodes for the searchers, we cannot use the same r6i.large instance [3]. Instead, we use three ultrawarm1.medium.search instances each with 2vCPU and 15.25GB of RAM. This is strictly more compute resources than the searcher setup for LogCloud.

All timing measurements are repeated five times. Error bars representing the standard deviation are shown when applicable.

5.1 Compression Performance

First, we compare the storage footprint of the different log management solutions in Figure 9. For reference, we also plot the size of the logs compressed with Zstd, which is used in both LogGrep and LogCloud [10]. We break down the storage footprint of LogCloud further to show the size of the Parquet files where we store a compressed copy of the raw logs and that of our index data structures. We make the following observations:

- As previously noted in [25, 31], OpenSearch achieves poor compression performance for logs. The indices takes up nearly as much space as the original uncompressed logs.
- As expected, LogGrep outperforms plain Zstd compression due to log-specific compression algorithms [31].
- The Parquet file sizes dominate the storage footprint for LogCloud in all cases. Since the Parquet files use Zstd compression, in almost all cases the Parquet file sizes is almost identical to the plain Zstd-compressed logs.
- Across all five datasets, LogCloud (Parquet + index) achieves 12.4x geomean lower storage footprint against OpenSearch and 2.7x larger storage footprint compared to LogGrep. The LogCloud index itself achieves 115x geomean lower storage footprint compared to OpenSearch and 3.5x lower compared to LogGrep.

The last observation raises two questions. The first: *why LogCloud opts to store a copy of the logs in Parquet files using Zstd compression?* We could simply use LogGrep instead of Zstd to compress the Parquet row groups, and then the Parquet sizes in Figure 9 will be the same as the LogGrep sizes instead of the Zstd size. We note that LogCloud index sizes is still much smaller than the compressed LogGrep sizes.

However, using LogGrep instead of Zstd will cause LogCloud to lose inter-operability with other SQL-based systems like SparkSQL and Trino, which will no longer be able to read the compressed logs. One intended mode of operation for LogCloud is to work in conjunction with Trino by transparently speeding up SQL queries with filters that involve substring search by first performing a keyword search and then rewriting the SQL to point SparkSQL/Trino to a set of row groups.

The second question: *why bother optimizing LogCloud index size when it is much smaller than the compressed logs either in Parquet or LogGrep?* The answer has to do with another usage scenario of LogCloud. Increasingly, SREs and security teams are storing raw logs in Parquet-based data lake formats like Delta Lake and Apache Iceberg, through tools like AWS Security Lake [28]. In this case, LogCloud can be configured to directly index the data lake as an "external index". In this case, the Parquet storage footprint is a given, and the index size is the sole storage footprint of LogCloud, which should be negligible compared to the data lake itself.

[8]In fact, the failure of adapting LogGrep for object storage constituted part of the original motivation for the design of LogCloud.

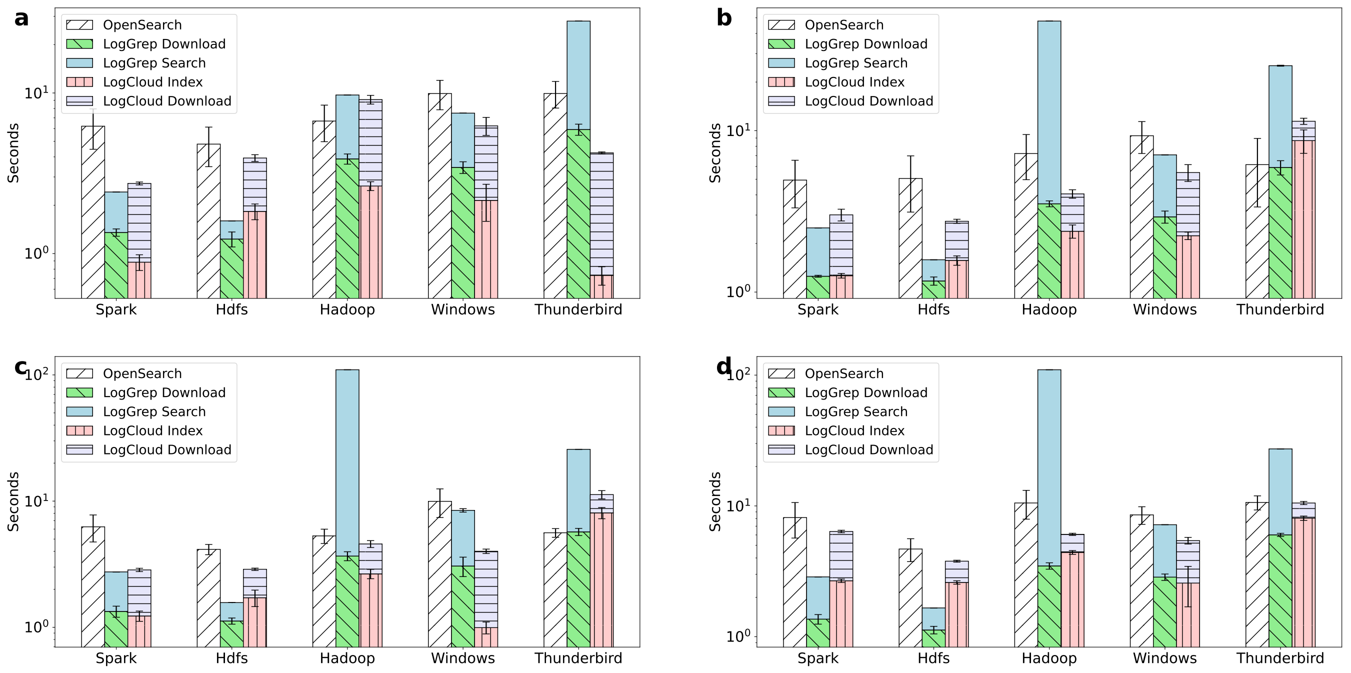

Figure 11: Search times for a) Common token b) Exact match on UUID c) Prefix match on UUID d) Substring match on UUID. For LogGrep we break down the runtime into the time it takes to download the compressed logs from S3 to disk and searching them. For LogCloud we also break down the runtime into searching the index and downloading the logs that matched.

5.1.1 ***Index Size Breakdown.*** Figure 10a shows the size breakdown of different components in LogCloud's index. As expected, in most cases, common tokens, templates and outliers are an order of magnitude or more smaller in size compared to the inverted index on the variable tokens. In most cases, the secondary BWT FM-index is larger than the compressed term dictionary and posting lists.

In the case of Windows logs, the BWT FM-index does not exist, because the number of variable tokens is very limited and does not hit the threshold to create the index. In fact, we see that the term dictionary size is much smaller than those of Hdfs and Spark, despite the uncompressed log size being more than 10x larger. In this case, all queries would brute-force search the term dictionaries.

Figure 10b further shows the sizes of the compressed term dictionaries, posting lists and the two main components of the BWT FM-index, the custom FM-index and the BWT mapping. These sizes reflect the optimizations on the size of the BWT mapping and custom posting list compression described in Section 2.3 and 3. We see that after these optimizations, the BWT mapping data structure still typically dominates the index size. In Section 5.3, we will show that without these optimizations, the BWT mapping and posting lists can get much larger.

5.2 Search Performance

In Figure 11 we show the search performance of the four 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 it takes 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 row groups.

On smaller datasets like Hdfs and Spark, LogGrep can perform very well. This is because the compressed logs can be quickly downloaded and scanned. However, on larger datasets like Hadoop, it can perform worse by an order of magnitude. In this case, the slow part is actually searching the downloaded compressed logs. We can see from Figure 10 that the Hadoop dataset has much more variable tokens than other datasets. Since LogGrep does not rely on indices it has to exhaustively scan all the variable tokens, causing bad performance.

OpenSearch performs well on most queries. However, since its FST indexing does not efficiently support substring queries, it is substantially slower on those compared to exact match and prefix queries. While it is around 30% slower on average across the five datasets, it is almost 2x slower on the Hadoop and Thunderbird datasets with large term dictionaries.

LogCloud performs comparable or better than OpenSearch in most cases (1.35x geomean speedup on UUID queries and 1.5x on common token). On the Hadoop dataset, LogCloud is faster than LogGrep by more than 15x for the UUID queries. The LogCloud search time can be divided into the time it takes to search the index dominated by latency-bound random accesses to object storage and the time it takes to download and filter the matching Parquet row groups, which is largely throughput bound. In most cases, the index search time typically dominates the time it takes to download and scan the matched row groups. This is the expected behavior

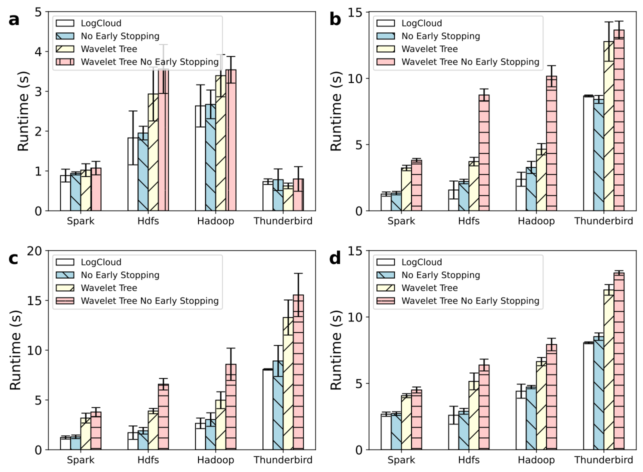

Figure 12: LogCloud index search times for a) Common token b) Exact match on UUID c) Prefix match on UUID d) Substring match on UUID on Spark, Hdfs, Hadoop and Thunderbird log datasets with and without early stopping, and with custom FM-index or wavelet trees.

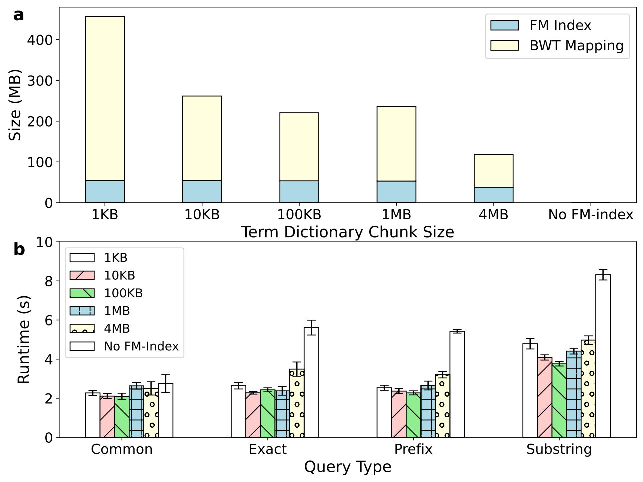

Figure 13: a) Term dictionary target chunk size vs index size. b) Query latency by type vs. term dictionary chunk size.

of an effective inverted index that limits the download to only row groups that contain possible hits.

5.3 Ablation Studies

LogCloud's main technical novelty lies in proposing an inverted index optimized for object storage by addressing the technical challenges listed in Section 2.3. As described in Section 3, LogCloud improves search speed through a custom FM-index implementation and early stopping, and lowers storage footprint through range reduction and custom posting list compression. In this section we validate the benefits of these optimizations.

5.3.1 ***FM-Index and Early Stopping.*** In Figure 12, we compare LogCloud's custom FM-index search time with a baseline wavelet tree implementation based on Algorithm 2, both with and without early stopping. The subsequent row group download speed is not compared as it is the same between the two strategies, which download the same row groups. We skip this analysis for the Windows dataset as no secondary index was constructed for this dataset due to the limited number of variable tokens.

We see that our custom FM-index performs much better (geomean 1.69x speedup) than the wavelet tree baseline in all scenarios except on common tokens for Spark and Thunderbird. In these cases, the secondary index is actually never searched as the query was short circuited after the common token matching shown in Figure 8. In terms of storage, the custom FM index is a bit smaller than the wavelet tree for all four datasets, with a 1.11x geomeon average size reduction.

In comparison, early stopping had a much smaller effect (geomean 1.10x speedup). Interestingly, early stopping applied to the wavelet tree implementation had a much bigger effect at 1.32x geomean speedup. Both early stopping and custom FM-index optimize just the secondary index traversal, which is only part of the total index access time. Since using the custom FM-index already accelerated this portion substantially, further optimizations have diminishing returns for the total index access time.

5.3.2 ***Range Reduction Trade-off.*** In Section 3, we describe the range reduction optimization to reduce the BWT mapping size. In short, instead of recording mappings to individual terms in the term dictionary, we record only mappings to chunks followed by an exhaustive scan of the chunk. Even though we still store the same number of integers as in the naive mapping, the dynamic range of each integer is now greatly reduced, making the mapping much more amenable to compression.

In Figure 13a we show for the Hadoop logs dataset, how the size of the BWT mapping change as a function of the term dictionary chunk size. We observe as the chunk size increases, the size of the BWT mapping decreases as the number of chunks, i.e. dynamic range of integers in the mapping, decrease. For reference, we see that the size of the FM-index itself stays roughly constant as expected. The BWT mapping at 1MB chunks is 45% of the size of the BWT mapping at 1KB chunks.

However, increasing the chunk size could lead to an increase in the search time, since the power of the secondary index is reduced as bigger chunks need to be downloaded and exhaustively scanned. In the extreme scenario, if the term dictionary is treated as a single chunk, then there is effectively no secondary index. We show the impact of increasing the chunk size on the index search times of the four query types in Figure 13b.

As the chunk size increases from 100KB to 1MB to 4MB, we see a gradual increase in the search time, though even at 4MB it still vastly outperforms the baseline without a secondary index. Interestingly, as the index size is reduced to 10KB and 1KB, the search time for some query types increase. This is because when reading these small chunks, the sizes of the chunks do not determine performance as much as the number of chunks to be read. If the FM-index points to two terms in a 100KB chunk, only one chunk

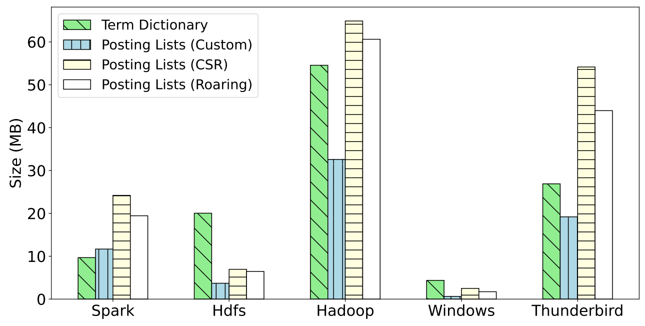

Figure 14: Comparing LogCloud's custom posting list compression with simple CSR format and Roaring Bitmaps.

has to be read if the chunk size is 100KB while two chunks need to be read with a chunk size of 1KB.

We note that there is a stable range between a chunk size of 10KB to 1MB where the BWT mapping size does not change too much. Such behavior is dependent on the nature of the logs being indexed. The chunk size is a tunable parameter in LogCloud, and can be optimized for specific log types.

5.3.3 ***Posting List Compression.*** In Section 3, we introduced a custom posting list compression method tailor-made for the variable tokens that we are indexing. Since LogCloud does not record posting lists for common tokens, and variable tokens like UUIDs typically exhibit strong temporal locality, most variable tokens we are indexing would end up in the same Parquet row group. As an example, we show in Figure 5 that most posting lists in the Hadoop and Thunderbird datasets indeed have length one.

In Figure 14, we compare our custom posting list compression against two baselines, a compressed CSR representation and Roaring Bitmaps [18]. In the CSR representation, we simply store the posting list values and the length of each posting list compressed by Zstd. We see that Roaring Bitmaps can improve storage footprint compared to this simple baseline, but our custom compression consistently performs much better across the five datasets (geomean 2.0x size reduction). Intuitively, this is because our representation only expends 1 bit for posting lists of size 1, while other representations need to encode "1" as an actual value.

Before our custom compression, the posting list was one of the bigger components in our index. With our custom compression, it becomes typically smaller than the term dictionary itself.

5.4 Summary

In summary, LogCloud is able to achieve faster search performance compared to OpenSearch UltraWarm, while having same order-of-magnitude storage footprint to LogGrep. Thanks to its custom inverted index, it can leverage cloud object stores like S3. This achieves the best of both worlds – fast search and cheap storage.

We perform ablation studies to validate that our custom FM-index and early stopping leads to a significant speedup in index search. We also validate that our range reduction techniques and custom posting list compression greatly reduce index size.

6 RELATED WORK

We discuss two categories of related work: (1) full text search engines commonly used for logs like ElasticSearch [7] and (2) search engines based on log-specific compression such as CLP, LogGrep and Loki [12, 25, 31].

ElasticSearch and its recent fork OpenSearch are general purpose text search engines commonly used to index and search production logs. However they lack log-specific indexing optimizations. As a result, they offer very poor compression ratios for logs compared to standard compression tools like gzip and Zstd [25, 31], which results in high storage cost for massive volumes of logs. In addition, ElasticSearch and OpenSearch require searcher VMs to have memory proportional to the index size. Larger index sizes translate directly into a higher memory requirement for searcher VMs, further increasing the cost.

The high storage cost of full-text indexing has led to alternative solutions that do away with the index, e.g. Grafana Loki, Scalyr or data lakes like Apache Iceberg [12, 14, 27, 28]. These solutions typically store logs compressed by standard algorithms such as Zstd [10], which results in a much smaller storage footprint compared to ElasticSearch/OpenSearch. To support efficient querying, these solutions preprocess logs to extract structured attributes to filter on with tools such as Vector or LogStash [9, 30]. This increases ingestion cost and operational burden, and users can only search indexed attributes.

Even full-text search on an indexed attribute could prove challenging. Loki famously suffers from poor performance when the labels have high cardinality, as is typically the case with important attributes like request IDs and Kubernetes pod names [26]. Even though the system no longer has to search through the surrounding log text, a substring search on billions of 21-character long Kubernetes pod names is still an expensive full text search problem. By doing away with the full text indices, these solutions simply push the cost from storage to querying.

Recent work such as CLP and LogGrep have sought to extract these structured attributes automatically [25, 31]. These works build on prior work on factoring logs into fixed templates and variable tokens based on pattern detection [32–34]. Repetitive text such as "block number" and "ARN" that appear inside the template is stored only once, which greatly improves the compression ratio. The variable tokens, which are the extracted attributes such as the actual block number or ARN, are then bucketed based on properties like length and alphabet composition. CLP and LogGrep can achieve over 2x better compression ratios compared to even highly optimized compression algorithms like Zstd, which is an order of magnitude better than the storage footprint of ElasticSearch [31]. Since templates are also stored, users are not limited to just searching indexed attributes.

However, these approaches do not ameliorate the problem of searching through the structured attributes that are extracted. Similar to Loki or data lake approaches, CLP and LogGrep do not build full text indices on these tokens, which are still exhaustively scanned during search. This can lead to high search cost for high cardinality tokens, especially when stored on object storage. Both the published CLP and LogGrep results are based on HDDs [25, 31].

7 DISCUSSION AND CONCLUSION

LogCloud was originally motivated by the failure to adapt LogGrep to object stores. It takes the straightforward approach of adding full text indices to the extracted structured attributes. However we soon ran into issues as existing indexing solutions either do not efficiently support substring search (FST, SSTable), or are poorly adapted to object storage (BWT, wavelet trees). The core technical contribution of LogCloud is the design of an object-storage-native full text index based on a custom BWT FM-index implementation. We show this index can be used effectively on the variable tokens extracted by LogGrep to perform fast and cheap substring searches against compressed logs on object storage.

LogCloud's object-storage optimizations are centered around simple observations on the performance of reads from object storage: reading 1MB often has the same performance as reading 100 bytes, and is generally more expensive than subsequent compute on the downloaded data.

This observation leads to the following approach to efficiently retrieve a small item from a large list of items: the large list should be stored in compressed chunks. The chunk containing the target item is located by position or index, downloaded, decompressed and searched. This approach is applied in LogCloud to the custom FM-index based on Jacobson's Rank, the chunked term dictionary and Parquet row groups, bringing numerous benefits:

- **Compression on the large list**, a key reason why our FM index has even better storage footprint than the wavelet tree FM-index which has built-in compression.
- **Range reduction** on the index pointing to the large list, responsible for the small BWT mapping size and short posting list lengths.
- **Fewer reads** if different small items lie in the same chunk: UUIDs like Kubernetes pod names commonly exhibit temporal locality in logs. Logs with the same UUID tend to be located in the same Parquet row group and can be retrieved with one read.

The target use case of LogCloud is human-in-the-loop analysis of near-line logs, e.g. for debugging production issues or tracking cybersecurity threats. It aims to have interactive search latency and very low storage cost for this use case. At the same time, we should list some non-objectives of LogCloud that are not as relevant for this use case:

- **Indexing latency**: it is not necessary to make ingested logs searchable in real-time, since we are targeting near-line instead of online logs. This means we can afford significant latency in log compression and index construction.
- **Exhaustive searches for common keywords** like "Error" will be very slow since LogCloud has to brute force search all the Parquet files. We note any kind of object-storage based log management solution would likely suffer from similar issues.
- **Millisecond search latency** similar to ElasticSearch hot tier is not a target of LogCloud. Since the end-user is a human instead of another service, latency SLAs can be relaxed to human-interactive levels, which means on the order of multiple seconds.
- **Online stream processing** of ingested logs to trigger alerts is not an optimization target of LogCloud. However, this functionality can be added to LogCloud by using a real-time query engine like Clickhouse on the unindexed logs in indexer memory.

We envision LogCloud to be used alongside existing log management solutions like ElasticSearch, Splunk or DataDog in practice [6, 7, 29]. Near-line logs can be offloaded to LogCloud to reduce cost while critical online log analytics remain on the existing solutions. LogCloud is implemented in around 4000 lines of C++. We have open-sourced LogCloud to facilitate further research.[9] Future work includes an efficient pipe-based query language like Splunk's SPL and better integration with other query engines like Trino.

8 APPENDIX

Table 1: All search queries

Dataset	Query Type	Query String
Spark	Common	ERROR
Hdfs	Common	error
Hadoop	Common	ERROR
Windows	Common	Error
Thunderbird	Exact	error
Spark	Exact	rdd_573_3
Hdfs	Exact	blk_5994635810173130289
Hadoop	Exact	blk_1076115144_2374320
Windows	Exact	1.1.7601.22667
Thunderbird	Exact	200512091831
Spark	Prefix	rdd_573
Hdfs	Prefix	blk_5994635810
Hadoop	Prefix	blk_1076115144
Windows	Prefix	1.1.7601
Thunderbird	Prefix	20051209183
Spark	Substring	573_3
Hdfs	Substring	5994635810
Hadoop	Substring	1076115144
Windows	Substring	7601.22667
Thunderbird	Substring	512091831

REFERENCES

[1] 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.
[2] 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
[3] Amazon Web Services, Inc. 2021. UltraWarm for Amazon OpenSearch Service (successor to Amazon Elasticsearch Service). https://docs.aws.amazon.com/opensearch-service/latest/developerguide/ultrawarm.html#ultrawarm-calc. Accessed: 2023-11-02.
[4] Andrzej Białecki, Robert Muir, Grant Ingersoll, and Lucid Imagination. 2012. Apache lucene 4. In *SIGIR 2012 workshop on open source information retrieval*. 17.
[5] Michael Burrows. 1994. A block-sorting lossless data compression algorithm. *SRS Research Report* 124 (1994).
[6] DataDog. 2023. *DataDog*. https://www.datadoghq.com/

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

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

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

[9] Elastic. 2023. *Logstash: Collect, Parse, Transform Logs*. https://www.elastic.co/logstash

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

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

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

[13] 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.

[14] Matano Inc. 2023. *The Open Source Security Lake Platform for AWS*. https://www.matano.dev/ Serverless, high scale, low cost, zero-ops security log analytics platform for AWS accounts, allowing for better security data management and analytics..

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

[16] 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.

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

[18] Daniel Lemire, Owen Kaser, Nathan Kurz, Luca Deri, Chris O'Hara, François Saint-Jacques, and Gregory Ssi-Yan-Kai. 2018. Roaring bitmaps: Implementation of an optimized software library. *Software: Practice and Experience* 48, 4 (2018), 867–895.

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

[20] 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. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 863–873.

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

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

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

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

[25] 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.

[26] sandstrom. 2018. *High cardinality labels*. https://github.com/grafana/loki/issues/91

[27] SentinelOne. 2023. DataSet. https://www.dataset.com/ Accessed: 2023-11-27.

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

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

[30] Vector. 2023. *A lightweight, ultra-fast tool for building observability pipelines*. https://vector.dev/

[31] 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.

[32] 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.

[33] Stephen Yang, Seo Jin Park, and John Ousterhout. 2018. {NanoLog}: A Nanosecond Scale Logging System. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 335–350.

[34] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. 2017. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 565–581.

[35] 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.