LittleLog: A Log Compression and Querying Service

Rahul Surti Shashank Saxena

Abstract

This paper presents LittleLog, a general-purpose log compression and query service that allows for permanent, lossless compressed storage while enabling queries directly on the compressed log files. LittleLog's motivation comes from the increased amount of data produced in recent years. With massive storage systems producing data, log files are being generated at a similar rate. To address this new development, LittleLog focuses on the compression and querying of log files produced by many of these systems.

Previous compression algorithms have impressive good compression ratios, but they do not enable querying on a compressed data space. This limitation requires log files (or any file in general) to be first uncompressed and then queried. **Our results show that we are able to query against log files with speeds up to 97% faster than `grep` occurrence count, 65% faster than `grep` line count while having compression ratios around 50% - all executing directly on compressed data.**

1 Introduction

This paper presents LittleLog, a log compression service that enables querying on a compressed data space.

Logs are produced during the execution of both user applications and components of a large system. With the ever increasing rate of data being produced and processed on a daily basis, the need for logs to account for each action and create a traceback is essential for any recovery system. Examples of this include web servers [15], data processing engines [19], databases[12], and distributed file systems[7, 17], among others.

Logs can come in many different formats. Some are private to certain systems whereas others are public and use popular formats such as CLF[13], NCSA, W3C and others. Rather than trying to tailor for specific log formats, LittleLog generalizes to work for any log format. To focus in on the compression, we leverage Succinct's algorithms and tailor optimizations and functionality for logs. Succinct [5] presents a method for compression and enables querying compressed data directly, yet it possesses some limitations.

To tailor for log files, LittleLog exposes an API that extends upon Succinct's API. Aside from being able to extract simply the offsets in a file as Succinct does, LittleLog extracts entire lines for a specific log file entry. This gives LittleLog the ability to simulate popular search tools for log files, such as `grep` [2] and `wc` [3].

For large log files, LittleLog utilizes a configurable parameter λ that is known as the shard size. This is used to specify how the input log file will be divided into shards of size roughly

λ. Different values of λ affect the shard compression ratio and overall query latency in different ways. This is analyzed in further detail in section 5. The introduction of λ is important for scalability. One of Succinct's downfalls is its inability to handle files larger than 2 GB.

To summarize, LittleLog contributes the following:

- We extend Succinct's algorithms and tailor querying on compressed data and apply it to a more realistic use case.
- LittleLog is configurable to specify a parameter λ that will shard an input file to chunks of size λ to prepare for compression and it accounts for scalability.

2 Motivation and Background

Log file entries are logically structured pieces of events that are divided into different fields usually delimited by linebreaks (newline characters). A field in a log entry describes information for a certain transaction such as a date, IP address, action performed, etc. A log file consists of many different log entries captured or generated by some system.

The primary motivation for this system is to be able to query logs with low latency and full accuracy while also reducing the overhead required to do log processing such as retrieval, storage, compression, and decompression. Tools used for log processing often require complex parsing algorithms and schemas to understand the type of logs that are being processed [8].

Today, data is being generated at an ever increasing rate. As systems become grow in complexity, the quantity of logs generated scales up [4] as well. An important goal of LittleLog is to minimize storage space and query latency in order to hasten the process of disaster recovery and gathering statistics on large-scale systems.

Research focusing on compression algorithms strives to generalize to any type of file. To the best of our knowledge, compression on log files is not heavily researched. Recent papers seem to focus on compressing the number of bytes per line using compression ideas based on text similarity [16, 10]. However, all these methods require decompression during querying and therefore contribute to increased query latency. Further, these algorithms do not account for text that is dissimilar and decompressing the text could result in errors based on the parsing scheme used.

LittleLog aims to focus on *true* permanent compression for log files. Specifically, we eliminate the need to decompress when querying log files. This reduces the extra overhead that many traditional querying schemes require [16].

3 Related Work

Our work relies on Succinct, a general-purpose compression algorithm. Succinct allows queries to be executed directly on a compressed data space. However, the query results that are returned are offsets within the file. While this is useful for locating data within the original log file, it does not provide the entire log entry which can be useful for disaster recovery or system tracebacks.

A naive approach to log compression and querying can be achieved through `gzip` and `grep`, which is analyzed in section 5. We see that multiple [6, 8, 20] compression algorithms exist for repeated data, but querying on these is infeasible due to the format of the compressed data.

Querying against compressed data does already exist, but the approach sacrifices some efficiency. This is best shown through the usage of `bzgrep` [1] on `bzip2` files. `bzgrep` allows querying on `bzip2` files, but the data is first decompressed and then fed into `grep`. This

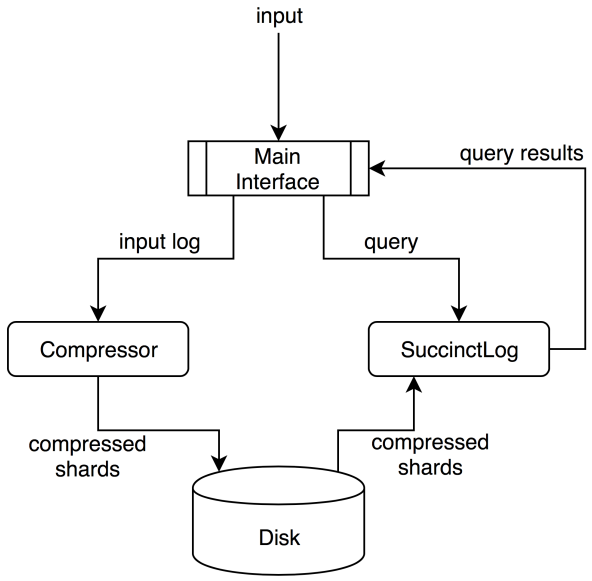

Figure 1: Architecture Design

incurs some extra overhead and resources may be wasted if querying is done constantly on the same compressed file.

4 Architecture

LittleLog is broken into 2 components: `compression` and `search`, which we expose in an API for ease of use.

4.1 API

```
f = compress(raw_input_log)
[l1, ...]  = search(f,query,limit)
occurrences = count(f,query,limit)
```

Figure 2: API exposed by LittleLog

LittleLog exposes an API that allows developers ease of use. The API exposes three methods: `compress`, `search`, and `count`.

The `compress` function takes in raw input log files and writes compressed, sharded `succinct` files to disk. In this step, we fully utilize Succinct's compression algorithm. Once compressed, there is no practical need to uncompress since querying can be done directly on the compressed log files, thus permanently saving storage space. The format of the compressed structure is further explained in section 4.2.

`search` is a powerful method that is borrowed from Succinct and further improved upon in LittleLog. Succinct's search function returns all offsets within the uncompressed file satisfying the query by executing the query against the compressed file. While this may be useful in some scenarios, it is limited for practical applications. LittleLog extends this by returning the entire line satisfying the query within the uncompressed file. This is arguably a more powerful and serves to be a more practical usage of querying since offsets within a file still require manual searching to extract any useful information. In addition, this simulates the behavior of industry-leading tools such as `grep`.

In order to prevent excessive terminal output, LittleLog offers an additional parameter of `limit` which acts as an output limiter. Specifically, the number of log entries given by `search` will be no greater than `limit`. By utilizing `limit`, we drastically reduce latency on queries when a large number of results are yielded while still returning relevant information about the query such as total number of matching results. Specific system performance is discussed in section 5.

The `count` method is analogous to UNIX's popular chaining of commands `grep <query> -o | wc -l`. We separate it here into a different method. The reason for doing this is because if simply the length of the total number of lines returned from `search` was given, we incur the overhead of finding the offsets within the file and extracting the line containing the offset. Rather, `count` employs a much faster mechanism to return the number of occurrences of the `query` within the file.

4.2 Design and Implementation

LittleLog is structurally divided into 2 sections: the compressor and the SuccinctLog, both accessed through the main interface. An input file is sent through the compressor where it is asynchronously sharded in-memory and compressed utilizing Succinct's compression algorithm. The shard size is given to the compressor via the user-specified parameter λ. The optimal value for λ is ideally as high as possible, but this is infeasible due to the high memory usage from Succinct's compression algorithm. This is actually the entire basis for the existence of λ. To potentially account for large files, sharding must be exercised [7].

Once a file has been sharded, compressed, and written to disk, it no longer needs to be decompressed as Succinct has enabled querying to be done directly on the compressed data. Queries are sent through the main interface where they are asynchronously executed using SuccinctLogs: an in-memory representation of the compressed log files. In our case, each SuccinctLog represents a shard of the original input log file. When a given query is being executed, it finds all offsets of the query within the compressed file and by maintaining an internal suffix array [11, 9]. Suffix arrays (like the one in Figure 3) themselves are an improvement over suffix trees [11, 18], which sometimes can take up more space than the string itself for operations such as substring searching. Taking this into account, there are internal suffix arrays stored within every SuccinctLog which transform strings (in our case, log entries) into a flattened array. This array is a sorted array that maps suffixes to their starting index. Succinct further cuts down on this by making their suffix arrays more space efficient at the cost of some extra computation. Rather than storing the entirety of the suffix within the suffix array, there is an index which is stored to find the location the subsequent character. Suffix arrays are therefore further compressed. This way, Succinct is able to give original offsets within the uncompressed file quickly by directly using the compressed file.

Suffix	i
$	7
a$	6
ana$	4
anana$	2
banana$	1
na$	5
nana$	3

Figure 3: Suffix array for the word "banana" sorted in ascending lexicographical order, courtesy of `https://en.wikipedia.org/wiki/Suffix_array`

LittleLog improves upon this by querying each shard asynchronously using multiple SuccinctLogs. A SuccinctLog loads a compressed file into memory before executing the query. Once executed, results are returned to the user in order of original uncompressed logfile, effectively simulating the Unix `grep` command. Result format depends on the which API method is called. `count` returns the total number of occurrences of the query in the original file whereas `search` returns full log entries satisfying the query. Because Succinct's underlying search algorithm returns just the offsets of a matching query in the original file, as it is generalized to any type of data, each log entry must be extracted individually. LittleLog utilizes Succinct's `extract` method for log entry regeneration. Leveraging the `limit` parameter allows LittleLog to avoid excessive regeneration and thus reduces overall latency. The log entry regeneration algorithm has a internal tunable parameter `shift`, which determines how many bytes to extract from the compressed file. The optimal shift value depends on the average length of log entries and can be tuned according to each input logfile.

Queries can be fed into both querying functions: `count` or `search`. Both have support for regular expressions. The former results in the number of occurrences of a regex string within every shard and aggregates the result. The latter returns the full log entry satisfying a query match given by the regular expression - similar to that of `grep`. Results are given back in an asynchronous manner. The `search` function leverages Succinct's `regex` matching capability described above, which returns offsets in the original uncompressed file, as well as Succinct's `extract` capability to generate the complete log entry, thus simulating the behavior of `grep`. It is recommended that logfiles maintain timestamps within the log entries so that results can be ordered if necessary via sorting. Most popular log formats (CLF, NCSA, Apache) already do this. The `count` method itself does not need to calculate the offsets of a query result, rather just the number of a given search will return, and thus offers low latency.

5 Experiments and Evaluation

The machine used to evaluate the performance of LittleLog was a virtual machine hosted on Microsoft Azure with 64 GB of memory and 8 vCPUs.

5.1 Methods

We first used a data generator to create a large log file of roughly 17.2 GB with log entries in the CLF format. We then proceed to shard and compress the same input file with varying λ. After manually pre-scanning the input file to gather a wide variety of queries, we run each query on each compressed file (represented by a set of compressed shards). We run queries using LittleLog and `UNIX grep` sequentially to ensure no resource competition and uniform testing environment while only manipulating λ. All queries are run for a total of 5 trials and the averages of these trials are shown.

5.2 Query Latency

We first tested using `search`. We find that search latency depends on the total number of results found. Reference Figure 4 and Figure 5. Queries with a very large results (high-yield queries), exhibited different behavior than queries with an order of magnitude less results (low-yield queries). Low-yield queries demonstrated improving performance as λ increases, whereas high-yield queries demonstrated the opposite. The improving performance for low-yield queries can be attributed to the decrease in overhead for loading compressed files into SuccinctLog's, as a smaller λ directly increases the number of shards an input log is split into. High-yield queries experiences latency multiple orders of magnitude larger than low-yield queries. Leveraging the `limit` parameter decreased latency by an order of magnitude, yet high-yield queries still face minute-level latencies. The bulk of this latency is due to Succinct's underlying search function, which calculates all offsets matching the query in the original file. This latency can be completely avoided by instead utilizing LittleLog's `count` method.

As `count` does not calculate the offset of the matching line nor generate the actual log entry, it experiences much less overhead and shows higher performance than that of `search`. We do not distinguish between high-yield and low-yield queries when utilizing `count` as all these queries demonstrate improving performance as λ increases. The largest shard size tested was 1GB. LittleLog retrieved the correct total count for a query with more than 40 million results in 2.8 seconds, whereas `UNIX grep` finished in 107.5 seconds; an improvement of over 97%. Even in the smallest possible shard size, LittleLog completes in 15.8 seconds, an 85% improvement. All trials of low-yield `search` queries (< 20 million) prove to be faster than

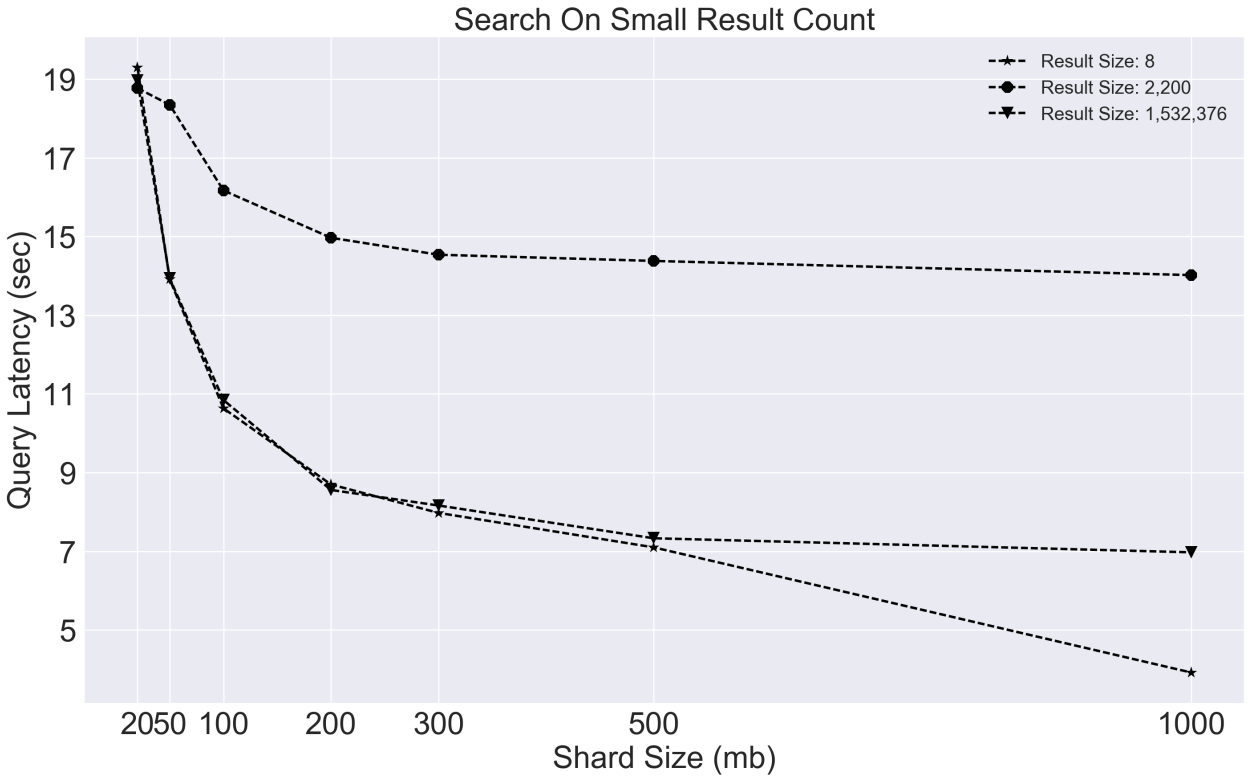

Figure 4: Search Query Latencies On Low Result Count

their `grep` equivalents.

Shard Size	Compressed Size
20 MB	8.5 GB
50 MB	8.7 GB
100 MB	8.9 GB
200 MB	9.1 GB
300 MB	9.2 GB
500 MB	9.3 GB
1 GB	9.4 GB

Table 1: Compressed Size Dependence on Shard Size. Original File Size 17.2 GB

5.3 Compression Ratio

Table 1 shows that the compressed files are roughly 50% of the input files. It includes the compressed size of the entire logfile after varying shard size. Our results show that as the shard size increases, the size of the compressed file also increases. We believe this is due to the combination of a wavelet tree[14] and compressed suffix arrays[18, 9]. Since there are less total strings being compressed per file, the space needed to store next line or subsequent string indices reduces. Roughly $O(n^2)$ bits are required for for a file with n characters. This agrees with the table mentioned above. Since n^2 does not grow linearly, it makes sense how smaller shard sizes result in a smaller compressed file size.

5.4 Analysis and Comparison

Given the varying query latencies and compression ratios, it's important to note a few key points.

For `search` queries alone, we categorize results as either high-yield or low-yield, as there is a significant difference in the performance

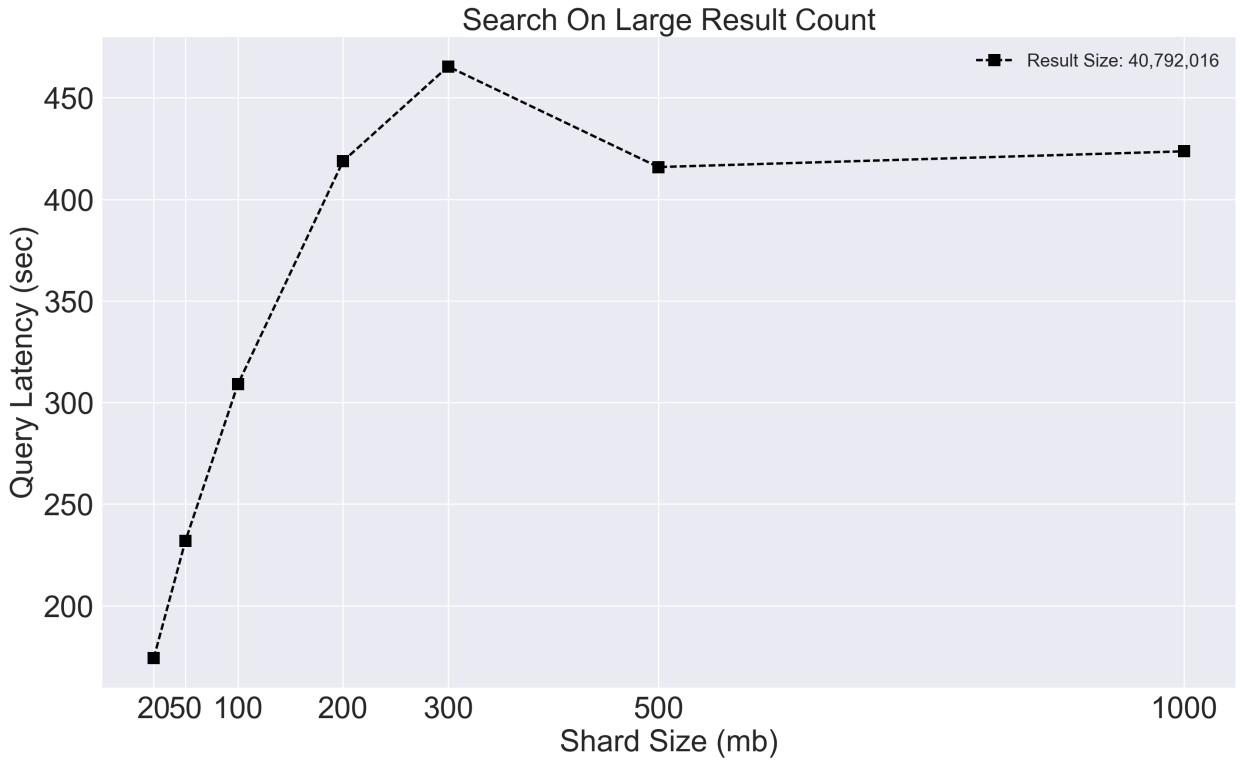

Figure 5: Search Query Latencies On High Result Count

between the two. As Succinct's underlying search algorithm only returns the offsets of a matching query in the original file, high-yield queries face extreme overhead calculating these offsets and regenerating complete log entries. Even after mitigating excessive log entry regeneration, these queries are not efficient enough to be used in practice. But by nature, they are infrequently used for analysis. Typically, useful query results tend to be very low-yield and are much easier to analyze. Even so, LittleLog offers a more performant `count` method to compensate for high-yield results. Although not directly considered querying, `count` can still be useful to know the statistics of certain query results in some cases. However, in all cases, LittleLog performs better than the native UNIX `grep`.

Although a smaller λ results in a smaller compressed file, this advantage in storage space is balanced with query latency. Figure 4 shows how, in general, query latency decreases as λ increases. This result is similar for the total occurrences of a certain string within the uncompressed file. This tradeoff is somewhat convenient to apply for practical systems. If a certain query latency is required for high-demanding systems, a specified λ value can be used. Vice-versa, if a certain compression ratio is desired at the cost of some latency, a known value of λ can be used. This value of λ can be calculate by recognizing that the graphs scale logarithmically.

6 Conclusion

6.1 Future Work and Limitations

Overall, LittleLog proves to be a viable system for log compression and querying. However, we still would like to discuss future options and work on LittleLog.

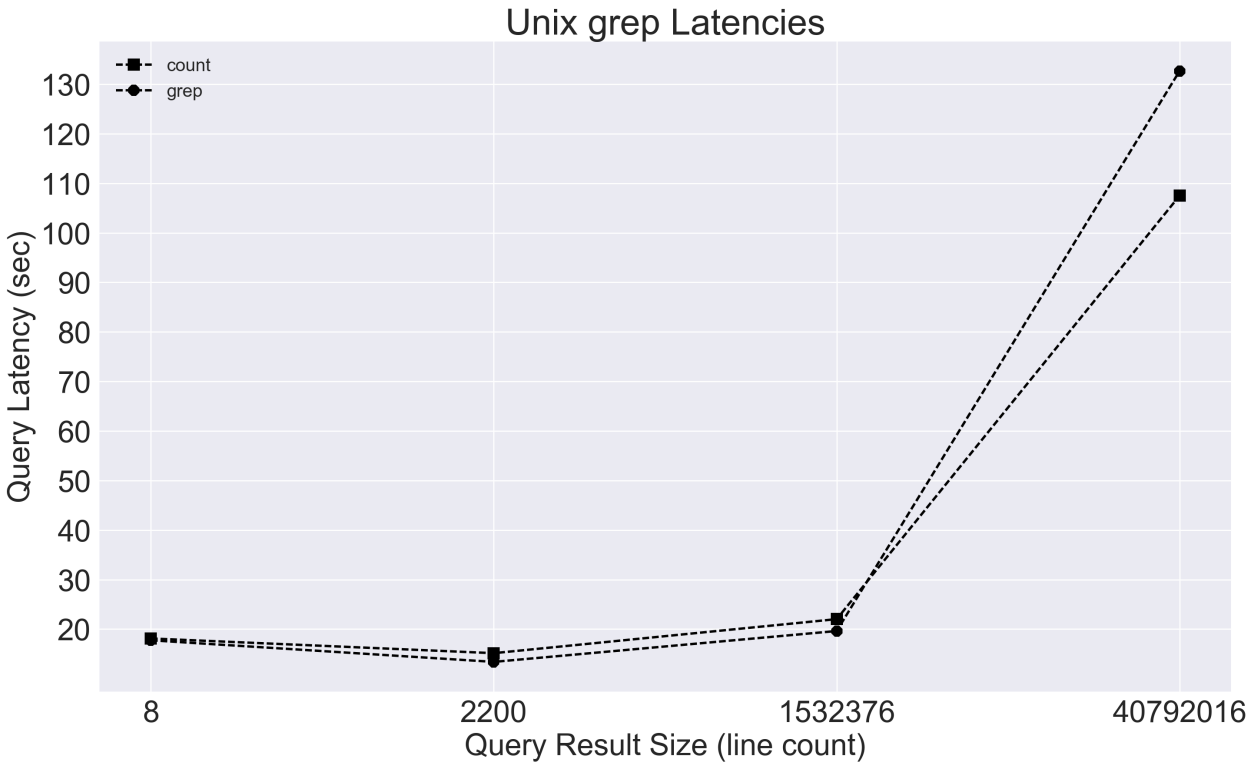

Figure 6: Unix Grep Query Latencies

Firstly, we would like to focus in on further compression. Currently, compression ratios for leading compression tools such as `gzip` or `bzip2` receive much better compression ratios. For our 17.2 GB file, `gzip` received an 81% compression ratio and `bzip2` received an astonishing 88% compression ratio. It would be interesting to try and see bottlenecks within the compression algorithm while striving to maintain direct querying possible.

Another area we tried to optimize on was memory usage. Currently, LittleLog is very heavy on memory usage due to sharding and buffer allocation. Our attempt at this was to include an extra parameter `limit` within the searching functions. However, this limits the total results that is returned, *not* the total results itself. Focusing in on the byte buffer allocations that LittleLog does in its underlying overhead calculations can drastically reduce memory usage. Cutting down on this usage could make LittleLog much more scalable.

As for practical implementations, we feel LittleLog can be a separate entity that is incorporated within large scale distributed file systems. File systems currently handle any type of file. However, seeing if one can be created or partitioned to solely take in logs would be interesting to note. Certain optimizations can be made since the nature of the files is known.

6.2 Summary

In this paper, we presented LittleLog: a general-purpose log compression and querying service that allows for permanent, lossless compressed storage while enabling queries directly on the compressed log files. LittleLog has support for query on compressed data directly with speeds faster than current industry leading tools such as UNIX `grep`, while also achieving competitive compression performance. Thus, Lit-

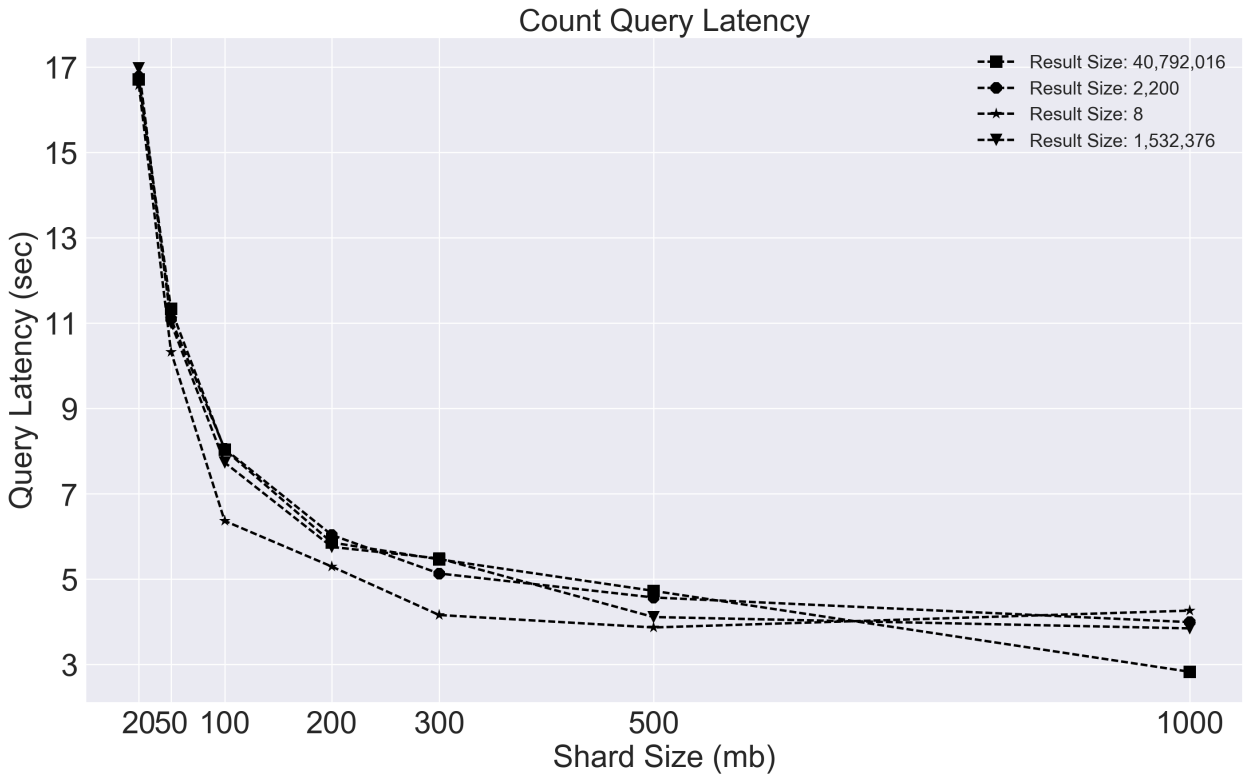

Figure 7: Count Query Latencies

tleLog presents itself to be a viable log compression and querying service. We hope to see it in practice some day.

References

[1] bzgrep, bzegrep, bzfgrep. https://www.mkssoftware.com/docs/man1/bzgrep.1.asp.

[2] Gnu grep. https://www.gnu.org/software/grep/manual/grep.pdf.

[3] wc. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/wc.html.

[4] The zettabyte era: Trends and analysis. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html, Aug 2017.

[5] AGARWAL, R., KHANDELWAL, A., AND STOICA, I. Succinct: Enabling queries on compressed data. In *NSDI* (2015), vol. 15, pp. 337–350.

[6] BALAKRISHNAN, R., AND SAHOO, R. K. Lossless compression for large scale cluster logs. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* (2006), IEEE, pp. 7–pp.

[7] BORTHAKUR, D., ET AL. Hdfs architecture guide. *Hadoop Apache Project 53* (2008).

[8] DEUTSCH, L. P. Gzip file format specification version 4.3.

[9] HALIM, S., AND HALIM, F. *Competitive Programming 3*. Lulu Independent Publish, 2013.

[10] HÄTÖNEN, K., BOULICAUT, J. F., KLEMETTINEN, M., MIETTINEN, M., AND MASSON, C. Comprehensive log compression with frequent patterns. In *Data Warehousing and Knowledge Discovery* (Berlin, Heidelberg, 2003), Y. Kambayashi, M. Mohania, and W. Wöß, Eds., Springer Berlin Heidelberg, pp. 360–370.

[11] KÄRKKÄINEN, J. Suffix cactus: A cross between suffix tree and suffix array. In *Combinatorial Pattern Matching* (Berlin, Heidelberg, 1995), Z. Galil and E. Ukkonen, Eds., Springer Berlin Heidelberg, pp. 191–204.

[12] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[13] LUOTONEN, A. The common log file format, 1995.

[14] NAVARRO, G. Wavelet trees for all. *Journal of Discrete Algorithms 25* (2014), 2–20.

[15] REESE, W. Nginx: the high-performance web server and reverse proxy. *Linux Journal 2008*, 173 (2008), 2.

[16] SKIBISKI, P., AND SWACHA, J. Fast and efficient log file compression. In *Local Proceedings of ADBIS 2007* (2007), pp. 56–69.

[17] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), USENIX Association, pp. 307–320.

[18] WEINER, P. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on* (1973), IEEE, pp. 1–11.

[19] ZAHARIA, M., XIN, R. S., WENDELL, P., DAS, T., ARMBRUST, M., DAVE, A., MENG, X., ROSEN, J., VENKATARAMAN, S., FRANKLIN, M. J., ET AL. Apache spark: a unified engine for big data processing. *Communications of the ACM 59*, 11 (2016), 56–65.

[20] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory 24*, 5 (1978), 530–536.