# Improving Compression of Massive Log Data

Robert Christensen
University of Utah
robertc@eng.utah.edu

May 1, 2013

**Abstract**

In this paper we explore a novel method of improving log compression by partitioning data into homogeneous buckets. By partitioning log records into buckets to improve homogeneous nature of log data, the effectiveness of generic compression methods, such as *bzip2* and *gzip*, can be improved.

A system is developed to archive log records using a `partition` function to distribute log records into buckets. Several `partition` functions are implemented and their improved compression ratio is reported when archiving several real world data sets. Several improvements to the method used in this paper are proposed.

## 1 Introduction

Log records are always being collected. In many instances the log is a collection of log records from a variety of sources and collected at a single location. System administrators use log data for a variety of tasks, such as identifying troubled hardware [8], analyzing network traffic [3], postmortem analysis of security breaches [2], and performance analysis [6]. Log data is a history of the status of a system. Because of the usefulness of this data, administrators are reluctant to delete this data, opting to archive the data in case it is needed in the future.

Each log record is typically a single line written in ASCII text, terminated with a new line character. The format of log data is designed for system administrators to easily read the log records as a massive text document without the need for special tools.

A system log for a large system could include multiple statuses intermingled, such as unusual memory issues with common DHCP status messages. Data compressors are more effective when the data being archived is consistent and predictable. Large log records lose consistency when events from different sources are intermingled into a single repository.

Splitting the log collection into multiple repositories, one repository for each log record type, is not an optimal solution because it would make system logging very difficult for administers of the system to find crucial information when needed. Records of one system fault may provide information about other system faults that would not necessarily be located in the same system log repository. Also, because a series of logs have a common source does not mean the log records are similar in structure. Current systems collect log data in a centralized source with a variety of heterogeneous records in close proximity.

A system is proposed to automatically distribute log data within an archive to improve the homogeneous nature of the archive, thus improving the effectiveness of using generic compressing utilities such as *bzip2* or *gzip*.

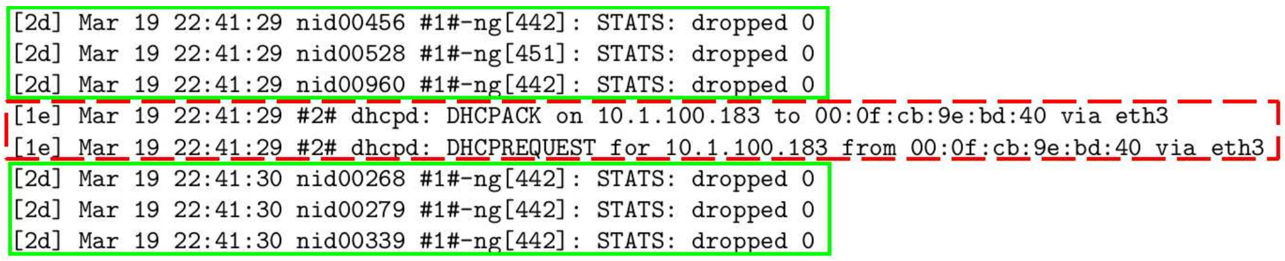

```
[2d] Mar 19 22:41:29 nid00456 #1#-ng[442]: STATS: dropped 0
[2d] Mar 19 22:41:29 nid00528 #1#-ng[451]: STATS: dropped 0
[2d] Mar 19 22:41:29 nid00960 #1#-ng[442]: STATS: dropped 0
[1e] Mar 19 22:41:29 #2# dhcpd: DHCPACK on 10.1.100.183 to 00:0f:cb:9e:bd:40 via eth3
[1e] Mar 19 22:41:29 #2# dhcpd: DHCPREQUEST for 10.1.100.183 from 00:0f:cb:9e:bd:40 via eth3
[2d] Mar 19 22:41:30 nid00268 #1#-ng[442]: STATS: dropped 0
[2d] Mar 19 22:41:30 nid00279 #1#-ng[442]: STATS: dropped 0
[2d] Mar 19 22:41:30 nid00339 #1#-ng[442]: STATS: dropped 0
```

Figure 1: Snapshot of Log data from Red Storm HPC system log

## 2 Problem Statement

A snapshot of a log file is shown in figure 1. This figure shows a segment of a system log from the HPC system Red Storm located at Sandia National Laboratory [6]. The characteristics of this segment is common in all real world log files studied. It is easily seen that a series of records of similar form are being recorded. For a small period, records of a different form are recorded by the system, creating a heterogenus log. The existence of these two inconsistent records, in a series of many records, decreases the consistency of the repository as a whole which hurts compressibility.

If the log records contained in the boxes boardered by the solid green line were group together and the two log records in the box boardered by the dashed red line were group together for archive, the archive would contain more homogenus data. Naturally, partitioning data into homogenus groups will decrease the randomness of the data. By doing this, compression algorithms can work more effectivly.

The system being proposed here has three basic steps:

1. *log record collection.* Log records are collected in order to be processed by the system. Log records can be collected and archived in real time but the option must exist to compress log records after the log is compleatly collected. Since it is common to delimit log records by a new line character, we will define a single log record as a string which is terminated with the new line character.

2. *log record partitioning.* The log archive consists of one or more buckets containing log records previously processed. The log partitioning step analyzes the new log record being inserted and determines which bucket to place the new record.

3. *log compression.* After the log records have been collected and partitioned the buckets containing the log records can be compressed. Each bucket is completely independent, so each bucket can be compressed separately in parallel. Also, since most compressing utilities compress data in blocks [4, 7] , when a bucket becomes sufficiently full, a block of data from the bucket can be compressed and stored.

In order to distribute log records in such a way that compression algorithms can work more effectively, we have defined the following formal statement of a system to partition log records.

Suppose there exists $n$ log records $l_1, l_2, \ldots, l_{n-1}, l_n$ and $m$ buckets $b_1, b_2, \ldots, b_{m-1}, b_m$. The buckets are ordered, meaning a log record $l_i$ can not be inserted into one of the buckets iff $\exists k$ s.t. $l_{i-1} \in b_k$ or $i = 1$. This ordering will allow for streaming algorithms to be used to distribute the records into the buckets.

We define a partitioning function `partition`$(l_i)$, which returns the index $k$ such that the record $l_i$ should be inserted as the next log record in the bucket $b_k$.

When a record is inserted into a bucket, the value of $k$ is recorded so the complete log can be reconstructed as it appeared in its original form when the buckets are unarchived. This collection of index value is also compressed and stored with the log archive.

## 3 Partition Functions

Each partitioning function implemented is explained below. The formal explanation of how it functions and a short description of unique behaviors of the partitioning function is also given, if applicable.

### 3.1 Segmentation

This distribution algorithm only considers the total number of log records the original log contains. Because of this, the technique only works if the number of log records is known beforehand.

When the log records are partitioned into $m$ partitions, the bucket $b_k$ will contain the series of log records:

$$\left[l_{\left(\frac{n}{m}\right)(i-1)}, l_{\left(\frac{n}{m}\right)(i-1)+1}, \ldots, l_{\left(\frac{n}{m}\right)i-2}, l_{\left(\frac{n}{m}\right)i-1}\right]$$

Partitioning the records this way will separate the log archive into $m$ segments, where each segment contains contiguous log records.

### 3.2 Round Robin

This distribution algorithm will insert the record into the buckets in a rotating way. Formally, this will return the bucket index $k = i \bmod m$. When distributing all log records each bucket will contain approximately the same number of records, similar to the Segmentation method. However, this method will shuffle log records in such a way that homogeneous data is often separated. Because of this, the bucketized log data will be more chaotic, thus causing the archive to be larger. This partitioning algorithm will show how poorly distributed data can significantly increase the total size of the output archive.

### 3.3 Edit Distance

This distribution algorithm uses the well known edit distance function to determine similarity between two log records. The function $ed(A, B)$ is defined in the following way. Given two strings of characters $A$ and $B$, the value of the edit distance is the minimum number of character modifications required to convert $A$ into $B$. A character modification can be one of three operations: character insertion, character deletion, or character substitution. The smaller the value of $ed(A, B)$, the more similar the character strings $A$ and $B$. The range of values returned from $ed(A, B)$ in $(0, \max(\text{len}(A), \text{len}(B)))$. The smallest value for when the strings $A$ and $B$ are exactly alike. The largest value for when the strings $A$ and $B$ have no similarity requiring every character in the string to be modified.

When inserting new records into the log archive, the edit distance function will be used to find the bucket with the most similarity. The bucket with the most similarity is the bucket the log record will be inserted into. The number of log records processed is expected to be large, so only a small sliding window of $s$ most recent log records will be compared in each bucket.

We will defined $S_k$ to be the sliding window records for the bucket $b_k$ with the constraint $0 \leq |S_k| \leq s$. The contents of $S_k$ include the last $s$ log entries inserted into $b_k$. If $b_k$ contains less then $s$ log records total, $S_k$ will contain all log records inserted into $b_k$. The notation $S_k[t]$ means the log record at the $t^{\text{th}}$ index if $S_k$ is being referenced.

When inserting the log record $l_i$, the bucket $b_k$ is being found such that we are finding the bucket $b_k$ that results in the smallest value for the formula:

$$\frac{\sum_{t=1}^{|S_k|} ed(l_i, S_k[t])}{|S_k|}$$

When the value of $k$ is found, the log record $l_i$ is then inserted into the bucket $b_k$.

If all the buckets containing at least one log record were had an average edit distance larger then some certain threshold and at least one bucket was empty, the log record being inserted will be inserted into the unused bucket. Otherwise, the log record will be inserted into the bucket with the smallest average edit distance, even if the value is very large.

For the implementation presented in this paper, the edit distance between the log record being inserted and the last $s$ log records inserted into each bucket was calculated for each log record inserted. Calculating the edit distance between two strings is a costly operation. This is because the process uses dynamic programming, where the operation is of the magnitude of $O(\text{len}(A)\text{len}(B))$. The real world data set used contained some log records over 100,000 characters. Even though most log records did not exceed 200 characters, the existence of an extremely large log record will result in extremely poor performance. The large log record must be inserted into the archive in order to guarantee lossless compression. The log records inserted after the large log record are likely to be much less unusual in size. Comparing a string with few characters with a string with many characters will always result in a large edit distance because a large number of insertion will always be required. Because of this, comparing any normal length string with a unusually large length string will result in a high edit distance. Because the bucket with the smallest average edit distance is where the new log record will be placed, the large log record will always be in the sliding window of some bucket because it will never be replaced. This causes significant performance degradation due to the time the edit distance function takes when given large strings.

The edit distance distribution algorithm was found to be extremely slow such that it was unusable for anything useful. In order to use this distribution scheme on the test data the process would take days to complete. Edit distance calculation optimizations could be used, but it was found that the edit distance similarity function did not perform as well as the other similarity functions explored. The edit distance distribution algorithm is included to show that it was considered as a distribution method, but performance both in time to compute and the resulting compression size showed that this is not a very good method.

## 3.4 Character Similarity

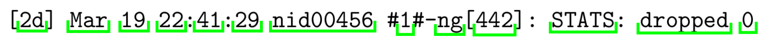
Each log record contains characters encoded as text. If two log records are homogeneous they should also have approximately the same count of similar characters. The Character Similarity partitioning function uses a distance function based on the number of appearances of each character in a log record.

Given the log records $l_a$ and $l_b$ which contain text. We define a character count array $C_a$ and $C_b$ for the log records $l_a$ and $l_b$. Suppose there is an index $x$ which is the binary representation of a character. For all $x$ we define $C_a[x]$ to be equal to the number of appearances the character whose binary respresentation is $x$ appears in the log record $l_a$. The character count array $C_b$ is also defined the same way using the log record $l_b$.

The character count distance between the log records $l_a$ and $l_b$ is defined using the formula:

$$\sum_{\forall x} |C_a[x] - C_b[x]|$$

If $l_a$ contains one more appearance of a character then $l_b$, that will contribute to increasing the character distance value by 1.



```
[2d] Mar 19 22:41:29 nid00456 #1#-ng[442]: STATS: dropped 0
```

Figure 2: Selection of words from a log record for Word Jaccard Distance

For explanation purposes, we will define a function $chardist(l_a, l_b)$ which will calculate the character count arrays for $l_a$ and $l_b$ and return the character count distance between the two log records. When implemented, the character count arrays should only be calculated once and stored in memory until they are no longer needed.

We will define $S_k$ to be the sliding window of records for the bucket $b_k$ with the constraint $0 \leq |S_k| \leq s$. The contents of $S_k$ include the last $s$ log entries inserted into the bucket $b_k$. If $|b_k| \leq s$, then $S_k$ will contain all log records inserted into $b_k$. The notation $S_k[t]$ means the log record at the $t^{\text{th}}$ index if $S_k$ is being referenced.

To find the optimal bucket $b_k$ to place a new log record $l_i$, the index $k$ is being found to minimized the formula:

$$\frac{\sum_{t=1}^{|S_k|} chardist(l_i, S_k[t])}{|S_k|}$$

### 3.5 Word Jaccard Distance

Using the well known set similarity Jaccard distance measure, this partitioning function selects which bucket the new log record should be placed. Given two sets $A$ and $B$, the Jaccard index value is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

The Jaccard index identifies similarity between the two sets. This value will be a rational number within the range $(0, 1)$, where the lower range identifies two sets completely different and the higher range identifies the two sets very similar. The Jaccard Distance, which identifies dissimilarity between two sets, is defined as $1 - J(A, B)$.

When a log record $l_i$ is going to be inserted into the log archive, a set of words found in the log record is created. We will use the symbol $W_i$ to be the set of words found in the log record $l_i$. Here, we define a word to be a continuous series of alphanumeric characters delimited by non-alphanumeric characters.

Figure 2 shows an example of the words that would be selected from a log record from the Red Storm HPC system log. Words may contain only letters, numbers, or a mixture of letters and numbers. When a log record is analyzed to create the word set, no attempt is made to determine the meaning of words. In the example, The time `22:41:29` is separated into three words. Similar log records appearing in the future will contain a different value in one or more of the time fields.

No context should be determined from the log record for fields such as this because no standard format exists to write time to a log record. The partitioning function becomes more generic by collecting all alpha numeric words in a log record.

We will define $S_k$ to be a sliding window of the last $s$ records inserted into the bucket $b_k$. The magnitude of elements in $S_k$ has the constraint $0 \leq |S_k| \leq s$. The value of $s$ is a parameter that can be provided to the archiver. If $S_k$ contains less then $s$ sets, $S_k$ will contains the set of words for all log records which have been inserted into $b_k$.

When inserting the log records $l_i$ into the log archive with its associated set of words $W_i$, the bucket $b_k$ is being found such that the the bucket $b_k$ results in the smallest value for the formula

$$\frac{\sum_{t=1}^{|S_k|} (1 - J(W_i, S_k[t]))}{|S_k|}$$

When the bucket $b_k$ is found, the log record $l_i$ is inserted into the bucket $b_k$ and $W_i$ is inserted into $S_k$, removing the oldest set in $S_k$ if needed.

If all buckets containing at least one log record had an average Jaccard distance larger then some certain threshold and at least one bucket was empty, the log record being inserted will be inserted into the unused bucket. If no empty buckets exist, the log record will be inserted into the bucket with the smallest average Jaccard distance even if the distance is extremely large.

## 3.6 $q$-gram Jaccard Distance

Using the same function as the Word Jaccard Distance, except rather then words for the log record being used to evaluate similarity all $q$-grams in a log record are used to evaluate similarity. A $q$-gram is a consecutive sequence of $q$ characters.

When a log record $l_i$ is going to be inserted into the log archive, all unique q-grams contained in $l_i$ are inserted into the set $W_i$.

This partitioning function is identical to the Word Jaccard Distance function described above except for the contents of the sets being compared. This method uses a set of all $q$-grams in the log record in place of complete words used in the Word Jaccard Distance method.

Even though speed of computation is not being explicitly evaluated in the experiments done, the $q$-gram Jaccard Distance method was found to take a very long time to complete. Even though it was not nearly as time consuming as the edit distance method, the total time taken was considered to be too slow to be of practical purposes.

The long computation time is not unexpected. For each log record inserted into the archive, a set will be created containing all $q$-grams from the log record. The number of times the similarity function is evaluated for each inserted log record is directly determined by the number of buckets and sliding window size. The values used for the number of buckets and sliding window size required each inserted log to be evaluated in the similarity function hundreds of times. Since the number of elements in the set was proportional to the number of characters in the set, evaluating the Jaccard distance could take a very long time.

## 3.7 Approximate $q$-gram Jaccard Distance

In order to significantly reduce computation time calculating Jaccard similarity between log records to find the appropriate bucket, The Jaccard similarity can be estimated using the KMV synopsis [1]. The KMV synopsis method allows for faster and more predictable computation time when calculating the Jaccard similarity between log records. This allows for log records containing many $q$-grams to easily be compared to other records without sacrificing time to archive while retaining the benefits of using $q$-grams.

All $q$-grams in the log record are hashed. The hash function used should be evenly distributed along the number line in the range desired. For the system developed in this paper the SHA1 hash function was used because the range of results was adequate, the speed of computing the hash of the strings was fast, and the randomness of the hash function for the purposes of this system were good. By assuming the hash function evenly distributes values along the number line, by taking the smallest $k$ hashes, we can approximate the total number of hashed values inserted onto the number line. The value of $k$ is a parameter provided to the algorithm.

Using similar techniques, intersection and union between two sets can be approximated while only storing a small number of hash values. The Jaccard similarity estimation can be very quickly calculated

| Source | Uncompressed (MB) | *bzip2* compressed (MB) | *gzip* compressed (MB) |
|---|---|---|---|
| Blue Gene/L | 509 | 35 | 45 |
| Thunderbird | 24,172 | 904 | 1511 |
| Red Storm | 32,596 | 664 | 1080 |
| Spirit (ICC2) | 28,583 | 389 | 666 |
| Liberty | 21,768 | 276 | 429 |
| Web Data | 7501 | 296 | 533 |

Table 1: Statistics for the size of the log data, both before and after compression.

| Source | log record count | median length | min length | max length |
|---|---|---|---|---|
| Blue Gene/L | 4,747,963 | 89 | 62 | 895 |
| Thunderbird | 211,212,192 | 89 | 22 | 1045 |
| Red Storm | 219,096,168 | 99 | 31 | 173,254 |
| Spirit (ICC2) | 272,298,969 | 100 | 13 | 994 |
| Liberty | 265,569,231 | 89 | 23 | 1037 |
| Web Data | 26,568,851 | 277 | 67 | 4187 |

Table 2: Statistics for the log record contained in each data source.

between two log records if a KMV synopsis has already been created for the log records.

This partitioning method uses the same process as that in section 3.6, except rather then building an exact set of Q-grams, a KMV synopsis is built and the value of the Jaccard distance is approximated.

## 4 Experiments

System logs from five supercomputers and one Apache web server log file was used to study the compression ratio between compressing using *bzip2* or *gzip* to compress the entire log without partitioning vs. partitioning using the various algorithms described in section 3.

Four of the supercomputer system logs come from computers located at Sandia National Laboratories: Thunderbird, Red Storm, Spirit (ICC2), and Liberty. The fifth supercomputer system log is from the Blue Gene/L computer located at Lawrence Livermore National Laboratory [6]. All supercomputer system logs have been scrubbed of personal data. Otherwise, the log data used in the experiments is the raw system log. When these system logs are referenced, they will be referenced by the name of the computer which it originated.

The Apache web server data came from the MesoWest project [5]. When this data is referenced, it will be called Web Data.

As we can see from table 1, Using *bzip2* or *gzip* to compress the log data is extremely effective. Compressing using *bzip2* is much more effective at compressing the log data used in this experiment. The *bzip2* compression algorithm takes significantly more time when compared to compressing using *gzip*. This is one of the primary reasons why *gzip* is used to compress large log files, as opposed to *bzip2* [3].

The archiving system developed allowed for many command line arguments to set different variables for a given run. These arguments include the partitioning algorithm used; arguments for the partitioning algorithm, such as maximum number of buckets; and arguments for the compression algorithm used internally, such as the block size for the compressor for each bucket.

All command line arguments were fixed except the variable being tested. Unless otherwise noted the following are the default values used in the experiment. The maximum number of buckets $m = 32$.

The sliding window depth $s = 10$. The $q$-gram size $q = 6$. The KMV synopsis value $k = 60$. If the value is not used because the partitioning algorithm does not need to value, the value is ignored. The partitioning algorithm is always explicitly listed for each experiment.

The $y$-axis scale shows the ratio:

$$\frac{\text{size of archive using the partitioning algorithm}}{\text{size of archive compressing the log whole}}$$

Figure 3 and figure 4 show that Round Robin performs very poorly. By distributing log records this way the buckets become more heterogeneous, resulting in data more difficult to compress. This shows that distributing log records among several buckets can result in worse compression if done incorrectly.

Edit distance partitioning resulted in worse performance when using the *bzip2* compressor in figure 3 and only slightly better performance when using the *gzip* compressor is Figure 4. Calculating the edit distance is a very costly operation. No further experimentation was performed using the edit distance partitioning function because it would take too long to be of any practical interest.

The word Jaccard distance partitioning algorithm performed better when the partitioner had more buckets available. Shown in figure 5 and figure 6, Under every condition, increasing the number of buckets improved the compression ratio. However, as the number of buckets increases the time required to process the data also increased because a log record being inserted into the archive must be compared to the last $s$ records in the bucket.

The improvement given by allowing for more buckets to be used by a partitioner depends on the partitioning algorithm used. While the word Jaccard distance partitioning algorithm improved greatly when more buckets were added, the character distance partitioning algorithm did not show significant gains with more buckets.

An experiment was done to evaluate the effects of choosing different kmv synopsis sizes when using the estimated $q$-gram Jaccard distance partitioning algorithm. The results are shown in figure 11 and figure 12. No significant differences exist within the results of this data, suggesting that within certain ranges the value of the kmv synopsis size does not significantly impact the effectiveness of the result.

**Quick review** of the experimental results show the most improvment in compression using the Jaccard $q$-gram partitioning algorithm, improving the compression using *bzip2* and *gzip* by 20% and 30% respectilvy. However, word Jaccard and approximate Jaccard performed almost as well.

## 5 Future Work

During the development of this log archiving system, several downfalls of the proposed system became apparent. Since some require changes to the method proposed here, they were not added to the current system. After these concerns are addressed, the log archiving system will be much more usable to administrators interested in long term log archives and will also have improved compression ratios over the results from this paper.

First, the number of buckets is fixed. In the current method, an unused bucket is used when the log record being inserted is not similar to any of the currently used buckets. After all buckets are used if a log record which is very different from the contents of every used bucket is going to be inserted into the archive, it will be placed in the bucket which most closely matches. However, it is easy to conceive of cases where doing this will decrease the homogeneous nature of the buckets.

Also, this limitation on how buckets are initially allocated can have significant repercussions. If the first log records inserted into the archive are very different from the log records to be inserted later, the buckets will be lopsided. Or, if one very unusual log record is the first inserted into a bucket it could be possible for that bucket to never be used again because the `partition` function will never select that bucket. This negatively impacts the compression ratio.

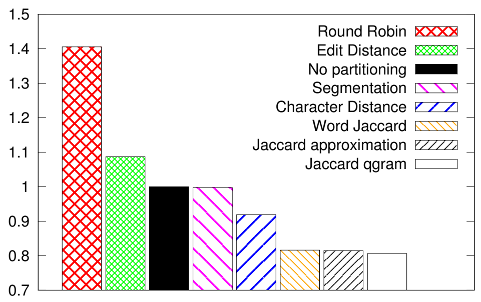


Figure 3: Vary partitioning algorithm on web data. *bzip2* compressor.

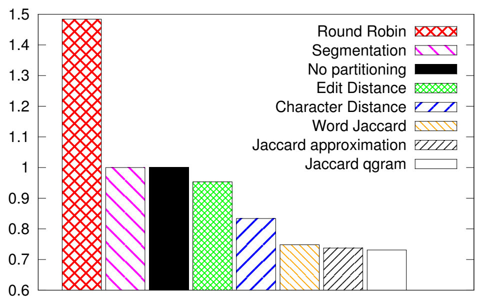


Figure 4: Vary partitioning algorithm on web data. *gzip* compressor.

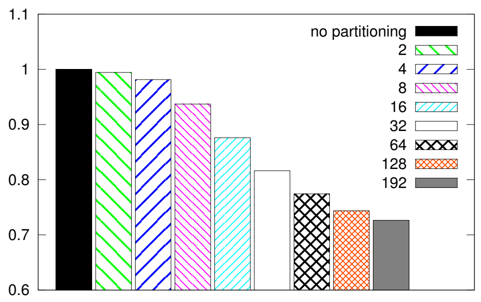


Figure 5: Vary number of buckets on web data using Word Jaccard Distance partitioning. *bzip2* compressor.

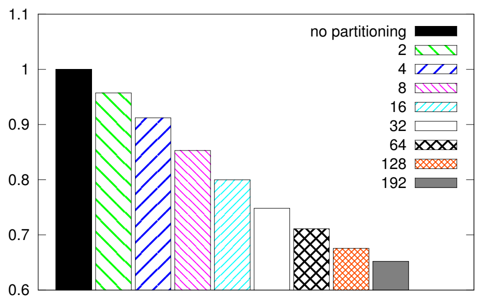


Figure 6: Vary number of buckets on web data using Word Jaccard Distance partitioning. *gzip* compressor.

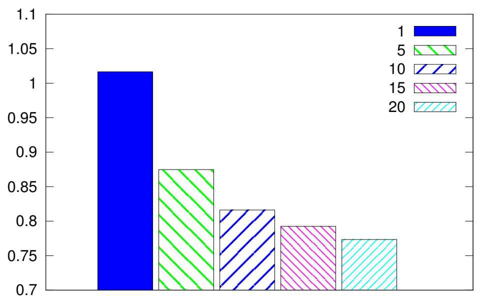


Figure 7: Vary sliding window depth on web data using Word Jaccard Distance partitioning. *bzip2* compressor.

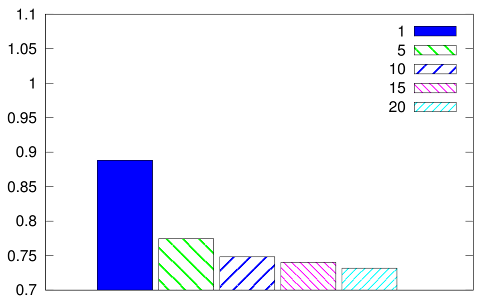


Figure 8: Vary sliding window depth on web data using Word Jaccard Distance partitioning. *gzip* compressor.

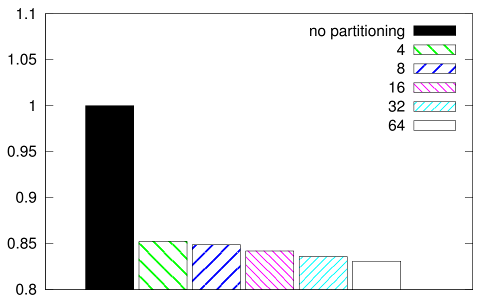


Figure 9: Vary number of buckets on Red Storm data using Character Distance partitioning. *bzip2* compressor.

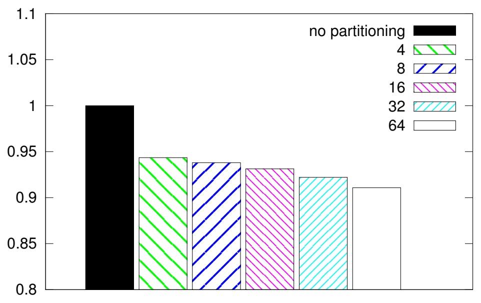


Figure 10: Vary number of buckets on Red Storm data using Character Distance partitioning. *gzip* compressor.

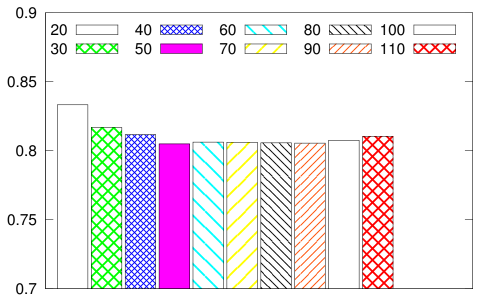


Figure 11: Vary kmv synopsis size on web data using Estimated Q-gram Jaccard Distance partitioning. *bzip2* compressor.

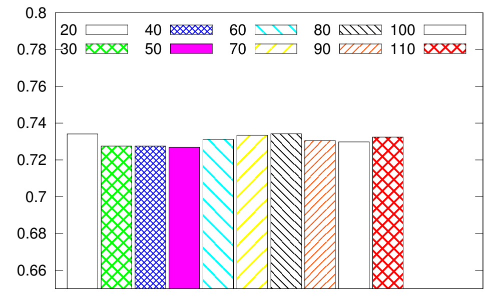


Figure 12: Vary kmv synopsis size on web data using Estimated Q-gram Jaccard Distance partitioning. *gzip* compressor.

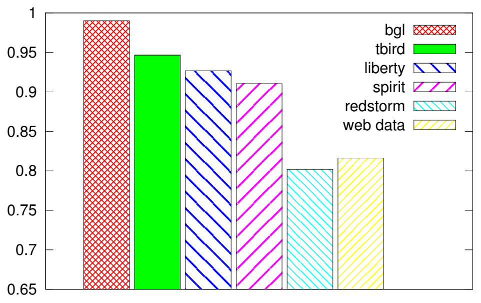


Figure 13: Vary data using Word Jaccard Distance partitioning. *bzip2* compressor.

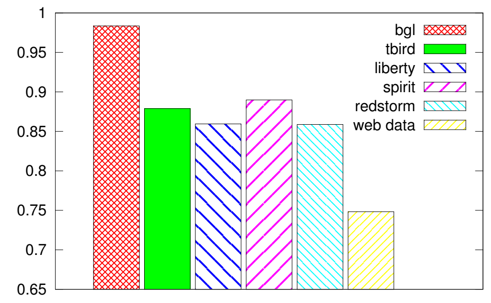


Figure 14: Vary data using Word Jaccard Distance partitioning. *gzip* compressor.

By observing trends in the contents of the buckets dynamically, it is possible to react to unusual log records in such a way that log records contained in a single bucket are very similar by creating and closing archive buckets when needed. This will decrease the negative impact of the first log records inserted into the archive as well.

Second, in all `partition` functions described in this paper which use a sliding window, the depth of the sliding window $s$ is fixed. The value of $s$ does impact the effectiveness of the archive, so larger values of $s$ are desired. However, if $s$ is larger the processing time to select the appropriate bucket increases.

If a bucket is extremely homogeneous, the value of $s$ does not need to be as large as a bucket which is not as homogeneous. By observing how homogeneous a bucket is, the value of $s$ can be adjusted dynamically to improve speed and improve the compression ratio of the log archive.

Some compression methods, such as *bzip2* and *gzip*, compress in blocks of data [7] [4]. The contents of one compression block should have no impact on a future block. Keeping a sliding window which straddles two compression blocks will not improve compression ratios, but it could negatively impact the compression ratio by restricting the contents of a bucket.

Third, if the algorithm were most closely tied to a specific compression algorithm additional optimizations could be applied. As already explained, by considering data in blocks of data rather then one continues bucket of many log records, the compression algorithm used would be better utilized.

Fourth, since log records are being placed in buckets based on the content of the log record, using a column based compression algorithm could significantly improve how efficiently the log archive can be compressed [9] [2]. The method used to distribute log records into buckets naturally inspires one to use such compression techniques rather then using generic compressors such as *bzip2* and *gzip*.

Fifth, in order for the log archive to be reviewed, it must be completely decompressed. Applying an index to the log archive could provide the ability to decompress specific sections. This would improve the usability of this method as an archiving tool because traditional log archiving methods do not support selective decompression of log archives [3].

Each of the future work topics discussed here are currently being studied and will be implemented in continued research.

## 6 Conclusion

In this paper we have implemented a preprocessing step for log archiving to be used in conjunction with traditional generic compression methods, such as *bzip2* and *gzip*. The preprocessing step separates log records using a `partition` function into homogeneous buckets. By improving the homogeneous nature of the log data, traditional compression methods are able to compress the log data more effectively.

Several `partition` functions were implemented using a variety of distance functions to detect similarity between log records. These functions include edit distance, word Jaccard distance, $q$-gram Jaccard distance, and $q$-gram Jaccard distance estimation. These functions were compared to each other and the traditional compression method of compressing the entire log using *bzip2* or *gzip*.

By using a `partition` function to improve the homogeneous nature of log data, log archives were able to be compressed more effectively then compressing the log file as a whole.

## References

[1] Kevin Beyer, Peter J Haas, Berthold Reinwald, Yannis Sismanis, and Rainer Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 199–210. ACM, 2007.

[2] Michail Vlachos Francesco Fusco, Marc Ph. Stoecklin. Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *VLDB*, 3(2), 2010.

[3] Francesco Fusco, Michail Vlachos, and Xenofontas Dimitropoulos. Rasterzip: compressing network monitoring data with support for partial decompression. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 51–64. ACM, 2012.

[4] Jean-Loup Gailly and Mark Adler. The gzip home page. *URL: http://www. gzip. org/(January 3, 2013)*, 1999.

[5] The University of Utah. Mesowest, January 2013.

[6] Adam Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 575–584. IEEE, 2007.

[7] Julian Seward. bzip2 and libbzip2. *avaliable at http://www. bzip. org*, 1996.

[8] Jon Stearley. Towards informatic analysis of syslogs. In *Cluster Computing, 2004 IEEE International Conference on*, pages 309–318. IEEE, 2004.

[9] Binh Dao Vo and Gurmeet Singh Manku. Radixzip: Linear time compression of token streams. In *Proceedings of the 33rd international conference on Very large data bases*, pages 1162–1172. VLDB Endowment, 2007.