Cowic: A Column-Wise Independent Compression for Log Stream Analysis

Hao Lin*, Jingyu Zhou†, Bin Yao†, Minyi Guo† and Jie Li†
*School of Software, Shanghai Jiao Tong University, Shanghai 200240, China
†Shanghai Key Laboratory of Scalable Computing and Systems, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai 200240, China

***Abstract*—Nowadays massive log streams are generated from many Internet and cloud services. Storing log streams consumes a large amount of disk space and incurs high cost. Traditional compression methods can be applied to reduce storage cost, but are inefficient for log analysis, because fetching relevant log entries from compressed data often requires retrieval and decompression of large blocks of data.**

We propose a column-wise compression approach for well-formatted log streams, where each log entry can be independently compressed or decompressed for analysis. Specifically, we separate a log entry into several columns and compress each column with different models. We have implemented our approach as a library and integrated it into two applications, a log search system and a log joining system. Experimental results show that our compression scheme outperforms traditional compression methods for decompression times and has a competitive compression ratio. For log search, our approach achieves better query times than using traditional compression algorithms for both in-core and out-of-core cases. For joining log streams, our approach achieves the same join quality with only 30% memory of uncompressed streams.

***Keywords*-Log Stream Compression, Log Search, Log Joining**

I. Introduction

Many Internet and cloud services generate a large amount of log data. It is estimated that semi-structured log data is produced at rates of 1-10 MB/s per machine in a typical data center hosting thousands of machines [1]. As a result, a modest 1000-node cluster could generate 86 TB log data each day. These data are not only valuable for service providers to predict market trends, to generate reports, and to discover fraud [2], [3], but also crucial for system administrators to analyze the root causes of bugs, to find performance bottlenecks, and to detect security issues [4], [1], [5].

To reduce the size of massive log data, compression is often employed before storing the data on disks, which can usually reduce storage requirements by a factor of 10. The problem with traditional compression schemes (e.g., gzip) is that they only work well when compressing the log in large blocks and cannot independently decompress a log entry. Thus retrieving a log entry requires to decompress a half block on average. For many log analysis applications, we often need to retrieve individual log entries, which results in reading many blocks of log data from disk and decompressing these blocks. This is inefficient because the majority of disk bandwidth and CPU cycles are wasted for unused log entries.

Corresponding author: Jingyu Zhou (jingyuzhou@sjtu.edu.cn).

Compression of log data at a record level is crucial for log analysis. With the ability of decompressing an entry independently, we can reduce the time for both fetching data from secondary storage and data decompression, thus effectively supporting many log analysis applications. However, such a record-level compression of log data faces two challenges. First, traditional compression techniques such as adaptive model and arithmetic code are unsuitable, because these techniques work inherently in a sequential fashion, exploiting redundancy within a short distance. It is difficult for a record-level compression algorithm to take advantage of duplicated information in nearby log entries. Second, each log entry is typically very short with little redundant information within the entry. Comparing to text document, log data can have a large vocabulary to be efficiently encoded. For these reasons, it is difficult for record-level compression algorithms to achieve high compression ratio.

We propose a column-wise compression method called Cowic for well-structured logs, supporting independent compression and decompression of individual log entries. Our method is based on a combination of semi-static dictionary model and Huffman code. Because a two pass approach is infeasible for log streams, our approach uses a small fraction of log as seed to train a compression model, assuming that the distributions of the seed and the whole dataset are similar. We take advantage of the structure of the log to construct a highly efficient compressing scheme. Specifically, we first split a log entry into several columns and build a model for each column. Because the data in the same column tend to share some common properties, a standalone model can fit the column better and do not interfere each other. Then for fresh words that do not appear in the seed, we construct an auxiliary word list in addition to the trained compression model. Finally, we further optimize our compression model with consideration of common phrases and columns of fixed format, e.g., timestamps or IP address.

We have implemented our Cowic compression as a library and integrated it into two applications, a log search system and a log joining system. Our evaluation with real world data shows that Cowic outperforms traditional block-level compression algorithms in decompression time and has a competitive compression ratio. For log search application, using Cowic is 3.6-71.1 times faster than gzip compression when data is in memory, and is 30.4-246.8% faster when data is on disk. For log stream joining application, using Cowic achieves the same join quality with only 30% memory of uncompressed streams.

This paper makes the following contributions:

- We propose a compression scheme for well-formatted logs, which allows each compressed log entry to be independently decompressed and is competitive to gzip in compression ratio. We have implemented our scheme in a C++ library [6] and made the code available to the public.
- We have integrated our compression scheme into a log search system and a log joining system. Our experimental results demonstrate that better query times can be achieved for the log search system, and the same join quality can be obtained with less memory.

The remainder of the paper is organized as follows. Section II summarizes related work. Section III illustrates two log analysis applications using our compression scheme. Section IV describes the details of our compression model and Section V illustrates the implementation of our library and two applications. Section VI evaluates the performance of our scheme with real world data. Finally, Section VII concludes our work.

II. Related Work

A. Compression Algorithms

Compression is a widely studied subject, where a model is built for the text to be compressed and is used to predict the next symbol. There are three types of compression models: *static model*, *semi-static model* and *adaptive model*. The static model uses the same model for all texts, which may perform badly when the text is different from the one used to build the model. For example, a model target at English text cannot efficiently compress a file written in Chinese. The semi-static model solves the above problem in two passes, where a specific model for the text is built at the first pass and the second pass uses the model for compression. The adaptive model [7] starts from a blank state and updates the model when meeting a new symbol. Many popular compression algorithms employ an adaptive dictionary scheme (e.g., LZ77 [8] and LZ78 [9]) to replace a substring in the text by a pointer to its previous occurrence. For instance, `gzip` is based on LZ77 and Unix utility `compress` is based on LZ78.

To compress log data for analysis applications, the above models are not efficient. Because log data come in streams, the two-pass approach of semi-static models is infeasible. For adaptive models, it is impossible to decode from an arbitrary position in the compressed text, because the code used for each symbol depends on both the initial configuration and previous text contents. Different from previous compression algorithms, our approach uses a small fraction of log stream to train a compression model, and then incrementally adds new words in an auxiliary list. Comparing to previous algorithms, our approach allows decompression of individual log entries in the compressed stream.

B. Log Compression and Analysis

Previous studies have targeted at the compression of different log data. Deorowicz and Grabowski [10] compress Apache log by splitting each log entry into several fields and build a model for each field. Balakrishnan and Sahoo [11] compress the event log of extreme-scale parallel machines. With the observation that most of the columns in adjacent records tend to be the same, they preprocess the log by a modified incremental encoding, which compares the given record with the previous record and encodes only the difference. Skibinski and Swacha [12] also observe that neighbor entries in logs are very similar in both structure and content. They replace the tokens of a new log entry with references to the previous one. All these studies first preprocess log data to reduce redundancy and then use a traditional compressor such as `gzip` as backend to compress the transformed data. The goal of all previous studies is to store the log for archive, not for log analysis purpose. When fetching a log entry from the compressed log data, they need to decompress the whole file. In contrast, our compression scheme targets at log analysis applications and supports efficient retrieval and decompression of individual log entries.

There are several solutions for building index and searching on log data. Splunk [15] is a commercial product that can store, index, query and visualize log data. Splunk stores raw data in a compressed form [17] and our evaluation with its free version shows that Splunk compresses data with `gzip` by default. ElasticSearch [16] is an open source project that stores data in compressed form using LZW compression [18]. To acquire a single log entry, both Splunk and ElasticSearch need to decompress a large data block. For well formatted log like Apache access log, our method can have a competitive compression ratio compared to `gzip` while outperforms it in decompression time, because our scheme can decompress a log entry independently.

Table I summarizes the features of existing work and our method. Our method compresses at record level and can independently decompress log entries, but requires a training stage and target at well structured log only. Table I also illustrates full-text retrieval systems with document compression. Zobel and Moffat [13] investigate how to add compression to an almost static corpus with a two-pass approach, which builds a dictionary model in the first pass and compresses in the second pass. For dynamic document repository, Moffat et al. [14] improve the document retrieval system by saving fresh words in an auxiliary lexicon. Our work differs from the full-text retrieval system in two aspects. We focus on the problem of compressing log streams where a two-pass approach is impossible. We target at well-structured logs and take advantage of the structure to improve the compression ratio.

TABLE I: A comparison of the features of related work.

Work	Compression Algorithm	Compression Unit	Independent Decompression	Training Required	Target
Cowic (this paper)	Dictionary + Huffman	Record	Y	Y	Structured Log
Deorowicz and Grabowski [10]	Preprocessing + General-purpose compressor (e.g., gzip)	File	N	N	Apache Log
Balakrishnan and Sahoo [11]				N	Supercomputer Event Log
Skibinski and Swacha[12]				N(Variant 1-3) Y(Variant 4-5)	Log
Full-text Retrieval System[13][14]	Dictionary + Huffman	File	Y	Y	Document
Splunk [15]	Gzip	Block	N	N	Log
ElasticSearch [16]	LZW	Block	N	N	Log

III. Application Scenarios

In this paper, we propose a record level compression scheme targeting at well formatted logs. Comparing to traditional compression schemes that usually compress data at block level, our scheme can directly decompress the log entry rather than decompress sequentially starting from beginning of a block, saving both decompression time and CPU resource. In the following, we discuss two applications using our compression scheme.

A. Index and Search on Log

The simplest and most common use of a debug log is to "grep" for a specific message [2]. However, this approach suffers from many limitations: 1) it is slow to scan the whole file for each query; and 2) it does not support range query and is difficult to build complex boolean queries. To address these limitations, a two-step process is often used, where we first create an index for the files and lookup the index during the search. To save disk space, the original data can be compressed.

Fig. 1(a) illustrates how to integrate block-level compression with procedures of building indexes and executing queries. An indexing program creates an inverted index and an ID map. For each keyword, the inverted index stores a list of log IDs associated with the keyword. The position of a specific log ID is stored in the ID map. The raw log stream is divided into blocks and compressed at the block level. When executing a query, we first search the index to obtain a list of log IDs. From the ID map, we can then obtain the offset for each log ID and calculate the block that stores this offset. Finally, we decompress the block and extract the log entry from the offset.

Fig. 1(b) shows how to integrate record-level compression with the above procedures. The only difference is that we store offsets of the compressed data in the ID map, instead of the offsets in the log stream. Then for each log ID, we can directly decompress the log entry starting from this offset in the compressed data. Comparing to block-level compression, our scheme can save decompression time and achieve faster query response (Section VI-D).

B. Join Log Streams

Our record-level compression can efficiently support equi-joining of compressed log streams. Fig. 2 illustrates the joining of two compressed streams $\mathbf{S1}$ and $\mathbf{S2}$. We first extract the join key by decompressing each entry and group other columns into compressed attributes. The join result is a concatenation of the join key and compressed attributes from both streams. Our record-level compression makes decompressing join result possible, because we can decompress each record independently with models of original compressed streams.

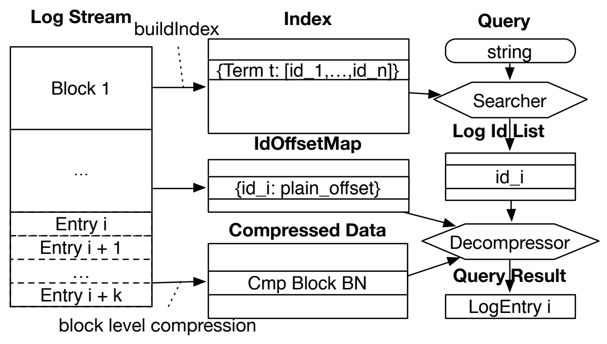
Our record-level compression can also improve join quality. The join operation is resource intensive because it needs to store and check unbounded states for two infinite input streams. To deal with the unbounded states, a common solution takes a sliding window approach [19], restricting the set of joining tuples to a bounded window size. For instance, consider an online auction service with the following two streams of auctions and bids:

$$\mathbf{S1} : \text{Auction(auction-id, seller-id, ...)}$$
$$\mathbf{S2} : \text{Bid(auction-id, bidder-id, ...)}$$

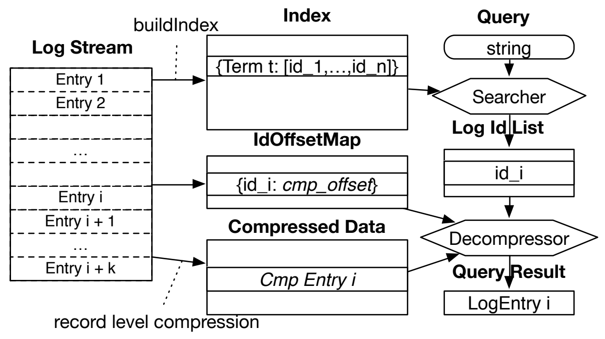
If we want to know the average number of bids for each seller received for his or her auctions during the last five days, we need to join stream $\mathbf{S1}$ and $\mathbf{S2}$. The sliding window size for $\mathbf{S1}$ is the maximum lifetime of an auction, and the window size for $\mathbf{S2}$ is the past five days. Even using the sliding window approach, the entries within the window may still exceed memory limit. As a result, load shedding is often used to provide approximate instead of accurate query result, such as random dropping [20], frequency-based dropping [21], and age-based dropping [22]. We can measure approximate join quality by the number of joined log entries. If we can compress log entries by a factor of N, then we can store N times log entries and improve the join quality, without increasing the memory usage (Section VI-E).

IV. Design

In this section, we first discuss the API of our compression library and its design choices, and then describe our compression model and optimizations.



(a) Block Level Compression



(b) Record Level Compression

Fig. 1: A comparison of block and record level compression in building index and querying logs.

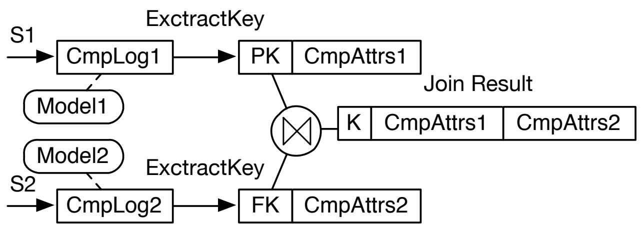


Fig. 2: The join process of two compressed streams **S1** and **S2**. PK, FK, and $CmpAttrs$ denote primary key, foreign key, and compressed attributes, respectively.

```
class Cowic{
  Cowic();
  void train (const string & seedFile, const string & modelFile);
  void loadModel(const string & modelFile);
   string compress(const string & entry);
   string decompress(const string & compressedEntry);
}
```

Fig. 3: The Cowic C++ library API.

```
Cowic compressor;
compressor.loadModel("example.mdl");
while (1) {
   string logEntry = getEntryFromStream();

   // assign log ID and update index
   int logId = getUniqueId();
  updateIndex(logEntry, logId);

   string binary = compressor.compress(logEntry);
   // store compressed data
   int offset = storeDataToFile(binary);

   // update ID map with location in compressed file
  updateIdMap(logId, offset, binary.size());
}
```

Fig. 4: A compression example of indexing a log stream.

A. API and Design Choices

To compress logs, a static model may perform poorly when we switch the log to be compressed, because the new log can have a different distribution from the compression model. On the other hand, a self-adaptive model can achieve a high compression ratio, but cannot decompress a single entry from a random place in the compressed data.

We choose to use a semi-static compression model, because it can achieve both goals of a reasonable compression ratio and independent decompression of a single log entry. Traditional semi-static model needs to scan the whole dataset to build a compression model and then compresses the data in a second pass. Because log stream can be infinite, our approach first uses a seed file to train a compression model, and then utilizes the model to compress the log data. Assuming the word distribution in the seed is close to the rest of the log stream, this approach can achieve our goals. We discuss how to update the model for fresh words in Section IV-C3.

Fig. 3 illustrates the C++ API of our Cowic library. For each type of log stream, we need to train it with a seed file and the generated compression model is saved to a file. With the trained model, we can compress a log entry into binary codes or decompress them back into the original entry.

Fig. 4 illustrates how to use the API to index a log stream. When a log entry comes in, we assign a unique ID for the entry and update the inverted indexing using the ID. Then the entry is compressed and saved to a file, where the offset in the file and the length of the compressed entry are saved in an ID map.

Fig. 5 illustrates how to use the API for search. When receiving a user query, we first search the inverted index to get a list of log IDs. Then for each log ID, we look up the ID map to obtain its offset and length. Finally, we fetch and decompress the log entries from the compressed file, and the results are presented to the user.

B. Basic Compression Model

Our compression model is based on Huffman code [23], which assigns shorter codes for words that appear more frequently in a dictionary. Specifically, Each non-alphanumerical character is treated as word delimiter, and continu-

```
Coiwc decompressor;
decompressor.loadModel("example.mdl");

while (1) {
  string query = getUserQuery();

  // get list of result ids
  vector<int> logIdList = queryByIndex(query);

  vector<string> logEntryList;
  for( int logId : logIdList ){
    // find location by id and extract the binary entry
    int offset , length;
    getOffsetById(logId, offset , length);
    string binary = getCompressEntry(offset , length);

    // decompress the entry and save it
    string plain = decompressor.decompress(binary);
    logEntryList .push_back(plain);
  }

  presentResult ( logEntryList );
}
```

Fig. 5: A decompression example of log search.

ous word delimiter is treated as a single word. Take `"http://www.martiworkshop.com/"` as an example. It will be separated into nine words ‘`"`’, ‘http’, ‘://’, ‘www’, ‘.’, ‘martiworkshop’, ‘.’, ‘com’, ‘`/"`’. Not every word needs to be added into the dictionary, and only those words whose frequency exceeds a threshold will be put into the dictionary. In this way, we can eliminate low frequency words like typos or self increasing keys. The advantage is reduced model size and there is little effect on the compression result.

Because the training seed is a small fraction of the whole data and we eliminate low frequency words, there are a number of fresh words during compression. We use a reserved mark code $\mathcal{F}$ to denote the occurrence of a fresh word, followed by an 8-bit integer to represent the word length and the word itself. Any word with a length larger than 255 will be separated into several words.

C. Optimizations

We perform a series of optimizations to improve the basic compression model, making it competitive to `gzip` in compression ratio with only 0.1% of training data. Because each log entry is compressed independently in our approach, common techniques using deltas between consecutive records are not applicable. Additionally, we need to pay special attention to fresh words that do not appear in the seed.

Our optimizations exploit the fact that structured logs consist of different types of columns, and construct different compression schemes for these columns.

1) Column-Wise Compression Model: We target at well structured log, where a log entry can be split into several fields (i.e., columns) of different types. Fig. 6 is an example of Apache access log, where the first column is visitor IP and the sixth column denotes a status code. These columns have different word distributions. For instance, IP addresses are dot separated numerical values ranging from 0 to 255, while status code is 3-digit value sparsely distributed from 100 to 505. Using a single dictionary cannot efficiently model all different columns. As a result, each column can have a separate compression model.

2) Phrase Model: Because certain sequences of words frequently appear in the log, it is more advantageous to treat them as one phrase and to assign a unique code for the phrase. For example, many URLs start with “http://www.” and end with “.com”. Treating these patterns as phrases, we can use a single code to represent the whole phrase rather than several codes, which can improve compression ratio and reduce decompression time.

We can also build phrases at the byte level rather than word level. I.e., each time we increase the phrase by one character. Compared to the word-level model, the byte-level model can find similar patterns in different words, e.g., prefix like “non”, “un” or suffix like “ing”, “action”. We choose to use the word-level model, because a common long phrase may only appear too few times in the seed to train the byte-level model and the word-level model converges faster for long phrases.

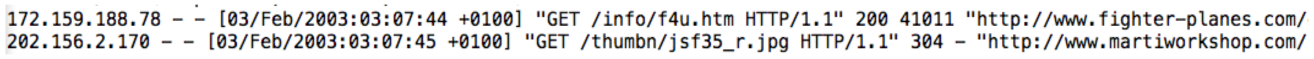
3) Auxiliary Word List Model: The pre-trained compression model may not fit well with new data — the incoming data can have frequent words that do not occur in the training data. Because we cannot modify the trained model, we create an additional auxiliary word list to efficiently encode these new frequent words. In the auxiliary word list, each word is encoded with a short code for all future occurrences. We append the auxiliary word list to the trained models so that together they can be used by the Cowic library for decompression.

Specifically, for each column, we maintain a map in memory to record the frequency of each fresh word. During compression, if the frequency of a fresh word exceeds a threshold, the word is saved in an auxiliary word list for future encodings. The difference with the basic model is that these words are marked with another unique code $\mathcal{W}$ in the compressed stream.

Assuming that a frequent word will appear earlier, we shall assign a shorter code to this word. We use Elias’s delta code [24] of the word’s index in the auxiliary list as the code. Such a coding scheme has the advantage that a small value integer has a short code.

4) Special Column Models: We define several special column models for those columns that do not fit into the dictionary model well, but can be efficiently compressed. The library user can provide hints so that Cowic may take advantages of these special models, which are discussed in the following.

Timestamp model. Timestamp does not fit into the dictionary model well, because the timestamps encountered during compressing stage differ from those during training stage. Additionally, the timestamp changes continuously and an auxiliary word list cannot model it well, either. Our timestamp model represents a timestamp by the UNIX time, which is defined as the number of seconds that have elapsed since 1970-07-01 00:00:00 UTC. To recover the original string, we



```
172.159.188.78 - - [03/Feb/2003:03:07:44 +0100] "GET /info/f4u.htm HTTP/1.1" 200 41011 "http://www.fighter-planes.com/
202.156.2.170 - - [03/Feb/2003:03:07:45 +0100] "GET /thumbn/jsf35_r.jpg HTTP/1.1" 304 - "http://www.martiworkshop.com/
```

Fig. 6: An example of Apache access log.

also save the timestamp format and timezone information in the model. A user needs to specify which column can be represented by timestamp model and configure the format correctly before training. Each timestamp is encoded as a 32-bit integer during compression. If most timestamps cannot be parsed correctly during training, the library reports a warning and chooses to use the dictionary model.

IP model. IP addresses are 32-bit numbers, usually stored in human-readable notations. IP addresses fit the dictionary model well only if a small fraction of them appears in the log. If the IP addresses spread in a wide range, e.g., web access log of millions of users, encoding the IP addresses in 32-bit numbers usually performs slightly better than the dictionary model.

Numerical Key Model. Numerical keys are commonly used to identify events in the log. Because a unique key value usually appears only once, the dictionary model cannot deal with numerical keys well. Instead, the user can specify the key column so that the key is represented by a 32-bit or 64-bit integer during compression.

5) Canonical Huffman Code: Decompression time is critical for fast query response during search, and we use a variant of Huffman code called Canonical Huffman Code [25] to reduce decompression time. The traditional Huffman code uses a tree structure to decode the compressed data. When meeting a code, we need to traverse the tree until reaching a leaf node, and then we can return the word this node represents. Traversing the tree is time consuming, because the algorithm needs to jump randomly through the tree structure for each bit of the code. Canonical Huffman Code eliminates the tree traversing during decompression by organizing codes such that longer codes (if selecting the most significant bits) have a numerically smaller value than shorter codes. Decompression is fast because we only need to treat the met bits as a number and compare it with the smallest code that has the same length. This requires an addition, a small table lookup, and a compare operation. Details on how to construct Canonical Huffman Code can be found in [26].

V. Implementation

We have implemented our approach in C++, supporting both Linux and Mac OS X platforms. The library consists of approximately 4,000 lines of C++ code measured by SLOCCount [27] and we have made it available on GitHub [6]. We have integrated our library into a log search system and a log joining system:

- **A log search system.** We have implemented a log indexing and searching system as illustrated in Fig. 1. Specifically, we use the open source Indri [28] tool for building inverted index and executing queries. Indri supports complex query language. For example, to find all visiting history for IP address 62.0.106.170, we can issue a query "#1(62 0 106 170)", which means the four terms appear in order. Using Indri, we can get a list of log IDs for each query. Then we can fetch the corresponding log entry for each log ID. For our record-level compression method, we use the ID map to find offsets in the compressed file and then decompress log entries from these offset directly. For block-level compression, we use the ID map to find offsets in the original log stream, then we decompress blocks storing these log entries to retrieve them. We use libarchive [29] to implement block-level compression, which provides streaming access for `gzip`, `zip`, and `LZW` compressed files. The implementation of the system consists of approximately 650 lines of C++ code.
- **A log joining system.** We modify RUBiS [30], an auction service modeled after eBay.com, to generate an open auction log and a bid log. We join these two logs using a sliding window approach and implement two different load shedding strategies, i.e., a time approach and a weight approach. The time approach sheds load by deleting the oldest auctions. The weight approach uses the number of join results of each auction as its weight, favoring the log entries with higher weights and deleting those with lower weights. The implementation of the system consists of approximately 800 lines of C++ code.

VI. Evaluation

This section evaluates the performance of our Cowic library with real world data. We try to answer the following questions:

- How does our compression scheme perform comparing to other algorithms?
- How does the Cowic library influence the search performance on logs?
- What impact does the Cowic library have on approximate log joins?

A. Datasets and Settings

We use three datasets in our evaluation: an Apache server log from Maximum Compression [31], a much larger Apache server log [32], and a supercomputer event log [33]. Table II lists the statistic information of these datasets. For training, only words occur two or more times are kept in the dictionary.

TABLE II: Statistic information of three datasets.

Dataset	Type	Size	Event Num
FP	Apache Access Log	20MB	109,481
Star	Apache Access Log	1.6GB	8,281,189
BGL	Supercomputer Event Log	718MB	4,747,963

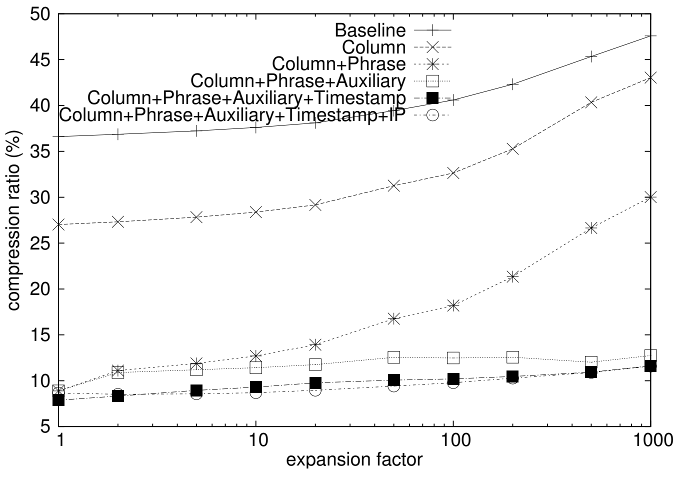


Fig. 7: A comparison of data compression ratio of different models for FP dataset.

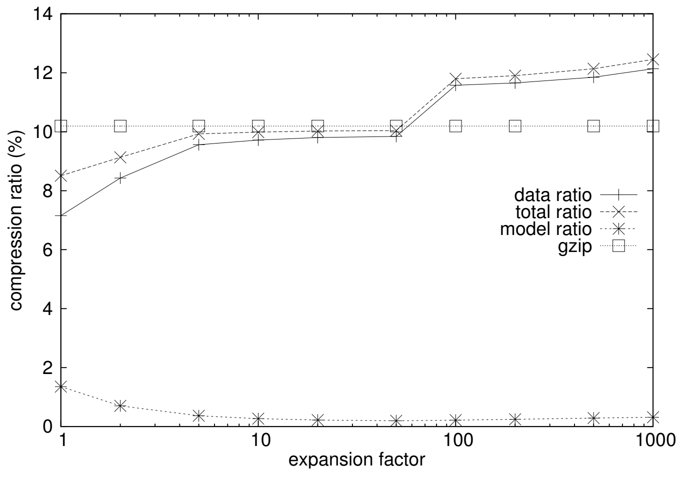


Fig. 8: Compression ratio for Star dataset under different expansion factors.

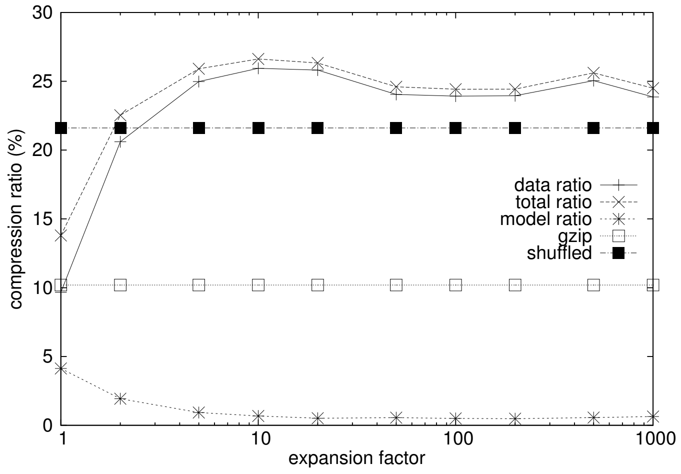


Fig. 9: Compression ratio for BGL dataset under different expansion factors.

Our test machine has a 1.6 GHz Intel Core i5-2400 CPU, 8 GB memory and a 7200 RPM disk, running Ubuntu Linux 12.04.

We use compression ratio to measure the effectiveness of compression algorithms:

$$compression\ ratio = \frac{Compressed\ Size}{Uncompressed\ Size}.$$

B. *Effectiveness of Our Compression Method*

We first study how our improvements work by integrating one improvement into our model each time and plot compression ratio against expansion factor. Expansion factor is defined as the ratio of the raw data size to the seed size. For example, an expansion factor of 1000 means the first 1/1000 of data are used as seed for training.

We compare different compression models over the FP dataset and Fig. 7 illustrates the results. The baseline is the basic compression model discussed in Section IV-B, which has the worst compression ratio. The proposed column wised model and phrase model help to improve compression ratio by 17.6-27.7%. With large expansion factors, the compression ratio degrades because less data are used for training our models. The auxiliary word list solves the above problem and improves compression ratio for large expansion factors up to 17.3%. Finally, the timestamp model and IP model further improve compression ratios by 0.23-3.14%. This experiment demonstrates the effectiveness of our compression optimizations.

Fig. 8 compares our approach using all optimizations with `gzip` compression for the Star dataset. For our approach, the compressed results include two parts, compressed data and the compression model. With larger expansion factors, the size of compressed data increases because less seed data are used to train precise compression models, and the model size decreases with smaller dictionary. When combing these two effects, the total compression ratio increases with expansion factors, but Cowic is still competitive to `gzip` — within 4% when we only use 0.1% data as seed.

Fig. 9 shows that compression ratios for the BGL dataset. `Gzip` has a higher compression ratio because there are many groups of events report similar errors, and `Gzip` uses an adaptive model to take advantage of this property, achieving better compression. In comparison, our compression model cannot capture similarity between consecutive log entries and has a higher compression ratio. If we shuffle the events of BGL dataset and compress it with `gzip` again (showed as shuffled in Fig. 9), the compression ratio is close to our scheme. This shows that our scheme is competitive in capturing redundancy information among log entries for compression.

C. *Training, Compression and Decompression Time*

This experiment studies the training time, compression time and decompression time for Star dataset under different expansion factors, and the results are illustrated in Fig. 10. We can observe that the training time decreases linearly with

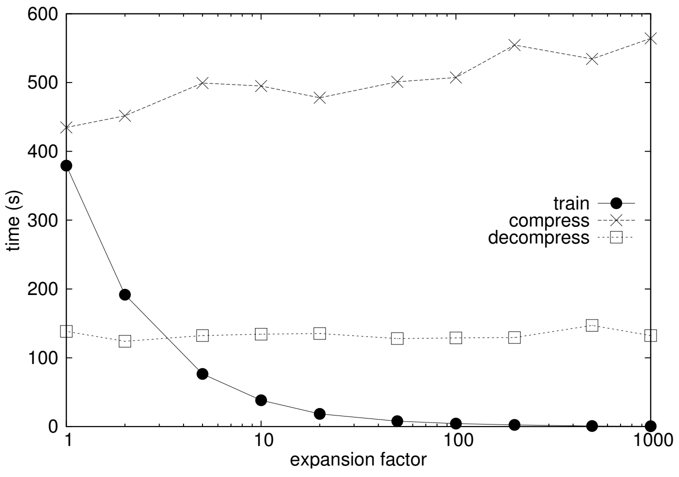


Fig. 10: The training, compression, and decompression time for Star dataset under different expansion factors.

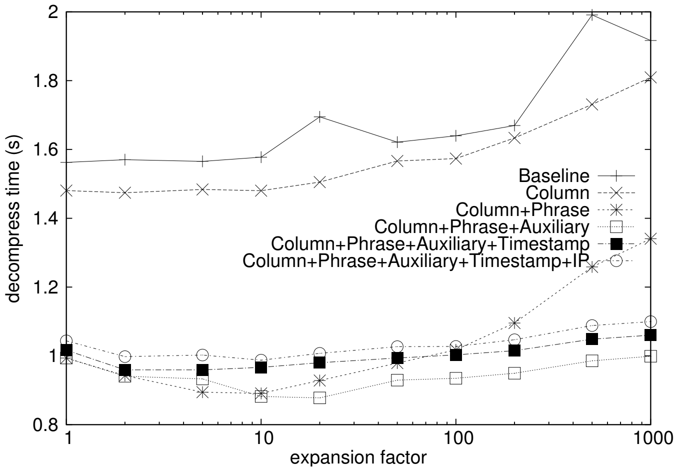


Fig. 11: Decompress times for FP dataset of different models when data are on disks.

expansion factors, because the training data is proportionally reduced with larger expansion factors. The compression time increases a little with larger expansion factors. This is because the compression algorithm needs to process the data to update the auxiliary word list of fresh words, which increases with larger expansion factors. The decompression time changes little with respect to expansion factors and is much smaller than compression time. When expansion factor is 1,000, compressing and decompressing a total of 8,281,189 entries cost 564s and 132s, respectively. In other words, we can compress and decompress at a speed of 14,683 entry/s and 62,736 entry/s, which is fast enough for common log analysis tasks.

Decompression time is most important since it influences query response time directly. We measure how each improvement in our model influences decompression times for FP dataset. Fig. 11 illustrates the results of decompression times for different approaches. The column-wise model, phrase model, and auxiliary word list can effectively reduce decompression time, because they can decrease the size of compressed data, resulting in less disk reading time. On the other hand, adding both timestamp model and IP model increases the complexity of decompression, and results in longer decompression times.

D. Impact on Query Speed

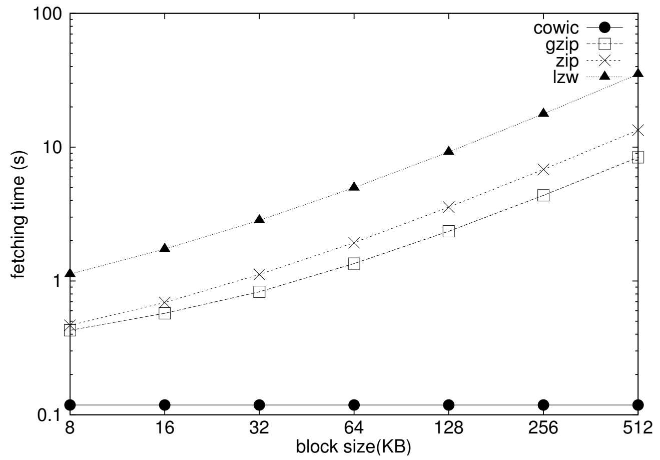
This experiment studies the impact of our compression library on searching logs. We compare the performance of three approaches: block-level compression of logs, our record-level compression of logs, and logs without compression. To perform a query on logs, the search system first parses the query and looks up an inverted index to obtain a list of relevant log IDs, and then fetches each log entry and decompresses it if necessary. In the experiment, we measure the time for fetching log entries and decompression, because parsing and lookup times are the same for different approaches.

We compare the performance of no compression, record-level compression and traditional block-level compression with 1,000 queries. For each query, we fetch the top 10 records returned by Indri [28] to form the first result page for the user. For the block level compression method, we vary the block size from 8 KB to 512 KB and try three different block-level compression algorithms, i.e., `gzip`, `zip` and `LZW`. For our method, we enable all compression models and take 0.1% data as seed.

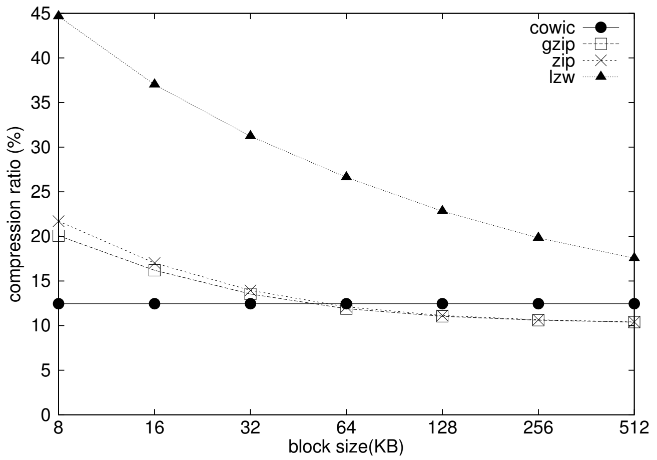
1) In-core Comparison: We first compare the performance when all data are in memory. Fig. 12 illustrates the results of fetching time and compression ratio for the Star dataset. From Fig. 12(a), we can observe that our Cowic compression is 3.6-71.1, 3.9-113.5, and 9.5-297.4 times faster than `gzip`, `zip`, and `LZW`, respectively. With increasing block sizes, these block-level compression algorithms become significantly more slower than our method. For instance, the decompression time increases to 8.4s (70 times slower than our method) if we choose 512 KB block size for `gzip`. This is because the decompression time increases linearly with block size, as on average we need to decompress a half block to fetch an entry, thus taking more time for larger blocks. In comparison, our approach allows independent decompression of log entries, thus decompressing less data than block-level compression algorithms.

Fig. 12(b) shows that the compression ratios of `gzip`, `zip` and `LZW` improve with a larger block size, because a more subtle model can be trained for a larger block. For instance, the compression ratio improves from only 20% to 10.4% when changing block size from 8 KB to 512 KB. The compression ratio of our Cowic approach is 12.5%, which is better than `gzip` and `zip` for small block sizes and outperforms `LZW` for all block sizes. Our method is more preferable because of much lower overhead of decompression time.

If no compression is used for the log data, then the fetching time is 0.026s, 22.0% of the Cowic time. However, no compression method requires seven times more memory space than our method.



(a) fetching time



(b) compression ratio

Fig. 12: A comparison of our record-level compression and traditional block-level compression in terms of compression ratio and fetching time when decompressing 10,000 entries. All data are in memory.

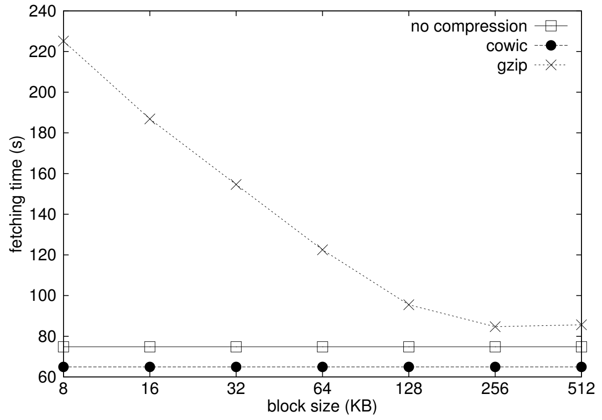


Fig. 13: An out-of-core comparison of no compression, Cowic and `gzip` when fetching 10,000 entries.

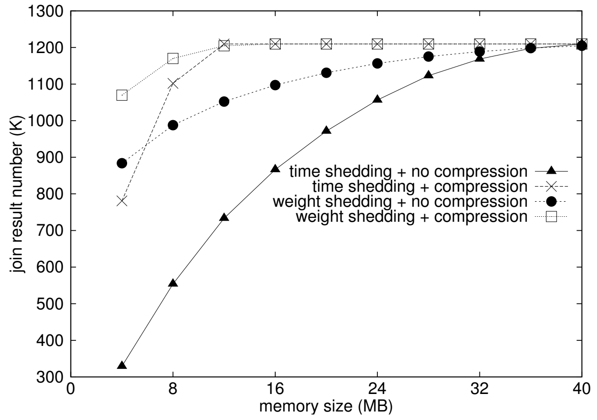


Fig. 14: A comparison of join quality for different shedding strategies and compression choices.

In conclusion, for well formatted logs our method can be at least 3.6 times faster than traditional compression algorithms with a possible cost of increasing compression ratio by less than 2.1%. Comparing to the uncompressed logs, our method reduces memory consumption by a factor of eight.

2) Out-of-core Comparison: This experiment compares the performance of different approaches when all data are on disk. We ensure the data are not in OS cache before the experiments, and the results are illustrated in Fig. 13. Our Cowic approach achieves the best performance of 64.9s and is better than the 74.8s no compression approach. This is because our approach reduces disk reading time, which dominates the fetching time. In fact, Cowic spends 64.6s on reading disk data and only 0.3s on decompressing data.

The block-level `gzip` compression performs the worst. The best fetching time for `gzip` is 84.6s with 256 KB block size, which is 30.4% slower than Cowic. The main reason for the inferior performance of `gzip` is that it needs to read half size of blocks on average from disk and to decompress the blocks. In contrast, Cowic reads less data from disk and decompresses less data. For `gzip`, the fetching time initially decreases with bigger block sizes because more effective compression reduces disk reading times. When the block size changes from 256 KB to 512 KB, the overhead of reading and decompressing blocks exceeds the benefits of compression. As a result, fetching time increases.

E. Impact on Approximate Join Quality

This experiment studies the impact of our compression method on log join quality of our modified RUBiS system. We vary the allocated memory for joining from 4 MB to 40 MB. In the experiment, the distribution of bids for auctions follows Zipf [34]. There are 1,209,600 log entries for each type of log. The size of auction log is 78 MB and the size of bid log is 77 MB. We use 1% of data as seed and after compression the log sizes are 18 MB and 25 MB, respectively.

Fig. 14 compares the join quality for both time and weight based load shedding strategies, where log streams are in raw format or in our compressed form. For both load shedding strategies, the compressed log streams perform better than raw log streams. To successfully join all log entries, the raw streams require 40 MB memory. In comparison, the compressed streams only need 12 MB (30%), because less memory is required for the same number of log entries. The weight shedding strategy performs better, because hot auctions that started in an early time will be dropped by the time shedding scheme. In contrast, these auctions are kept by the weight shedding strategy.

VII. Conclusions

This paper presents a compression scheme targeting at well-formatted log streams. We take advantage of the structure of the log and build a separate model for each column. Our scheme has the advantage that every log entry is independently decompressable, which makes decompressing an entry much faster than block-level compression schemes. We have integrated our compression scheme into a log search system and a log joining system. Our evaluation shows that our compression scheme is competitive to gzip in compression ratio for well-formatted logs, and outperforms traditional compression methods for decompression times. For log search application, our scheme is 3.6-71.1 times faster than gzip compression when data is in memory, and is 30.4%-246.8% faster when data is on disk. For log joining application, our scheme consumes only 30% memory for achieving the same join quality of uncompressed streams.

Acknowledgment

We thank anonymous referees for their helpful comments. This work was partially supported by the National Basic Research 973 Program of China (No. 2015CB352403), Shanghai Excellent Academic Leaders Plan (No. 11XD1402900), Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT), and NSFC (Grant No. 61272099, 61261160502, 61202025, 61428204).

References

[1] D. Logothetis, C. Trezzo, K. C. Webb, and K. Yocum, "In-situ MapReduce for Log Processing," in *Proceedings of USENIX Annual Technical Conference*, Portland, OR, 2011, pp. 115–129.

[2] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Communications of the ACM*, vol. 55, no. 2, pp. 55–61, 2012.

[3] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman, "Photon: Fault-tolerant and scalable joining of continuous data streams," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2013, pp. 577–588.

[4] Q. Fu, J.-G. Lou, Y. Wang, and J. Li, "Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis," in *Proceedings of the IEEE International Conference on Data Mining (ICDE)*, Miami, FL, Dec. 2009, pp. 149–158.

[5] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 117–132.

[6] "Cowic," 2014, available at https://github.com/linhao1990/cowic.git.

[7] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, no. 4, pp. 320–330, 1986.

[8] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[9] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[10] S. Deorowicz and S. Grabowski, "Sub-atomic field processing for improved web log compression," in *Proceedings of International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science*, 2008, pp. 551–556.

[11] R. Balakrishnan and R. K. Sahoo, "Lossless compression for large scale cluster logs," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006, pp. 7–7.

[12] P. Skibiński and J. Swacha, "Fast and efficient log file compression," in *Proceedings of 11th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, 2007, pp. 56–69.

[13] J. Zobel and A. Moffat, "Adding compression to a full-text retrieval system," *Software: Practice and Experience*, vol. 25, no. 8, pp. 891–903, 1995.

[14] A. Moffat, J. Zobel, and N. Sharman, "Text compression for dynamic document databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 9, no. 2, pp. 302–313, 1997.

[15] "Splunk," 2014, available at http://www.splunk.com.

[16] "ElasticSearch Storing Strategy," 2014, available at http://www.elasticsearch.org/guide/en/elasticsearch/reference/0.90/index-modules-store.html.

[17] "How Splunk Store Indexes," 2014, available at http://docs.splunk.com/Documentation/Splunk/latest/Indexer/HowSplunkstoresindexes.

[18] T. A. Welch, "A technique for high-performance data compression," *Computer*, vol. 17, no. 6, pp. 8–19, 1984.

[19] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2002, pp. 1–16.

[20] J. Kang, J. F. Naughton, and S. D. Viglas, "Evaluating window joins over unbounded streams," in *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, 2003, pp. 341–352.

[21] A. Das, J. Gehrke, and M. Riedewald, "Approximate join processing over data streams," in *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2003, pp. 40–51.

[22] U. Srivastava and J. Widom, "Memory-limited execution of windowed stream joins," in *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment, 2004, pp. 324–335.

[23] D. A. Huffman *et al.*, "A method for the construction of minimum redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[24] P. Elias, "Universal codeword sets and representations of the integers," *IEEE Transactions on Information Theory*, vol. 21, no. 2, pp. 194–203, 1975.

[25] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Communications of the ACM*, vol. 7, no. 3, pp. 166–169, 1964.

[26] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999.

[27] D. A. Wheeler, "Sloccount," 2001, available at http://www.dwheeler.com/sloccount/.

[28] "Indri," 2013, available at http://www.lemurproject.org/indri/.

[29] "Libarchive," 2014, available at http://www.libarchive.org.

[30] C. Amza, E. Cecchet, A. Chanda, S. Elnikety, A. Cox, R. Gil, J. Marguerite, K. Rajamani, and W. Zwaenepoel, "Bottleneck characterization of dynamic web site benchmarks," *Technical Report TR-02-391, Rice University*, 2002.

[31] "MaximumCompression," 2011, available at http://www.maximumcompression.com/.

[32] "Star Wars Kid Apache Log," 2014, available at http://waxy.org/2008/05/star_wars_kid_the_data_dump/.

[33] A. Oliner and J. Stearley, "What supercomputers say: A study of five system logs," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 575–584.

[34] G. K. Zipf, *Human behavior and the principle of least effort*. Addison-Wesley Press, 1949.