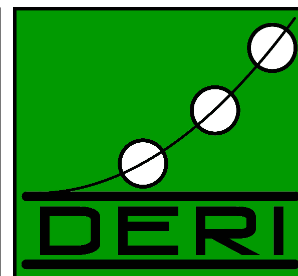
DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE




# Adaptive Frame of Reference for Compressing Inverted Lists

Renaud Delbru
Stephane Campinas

Krystian Samp
Giovanni Tummarello

DERI Technical Report 2010-12-16

December 2010

DERI – DIGITAL ENTERPRISE RESEARCH INSTITUTE

**DERI Galway**
IDA Business Park
Lower Dangan
Galway, Ireland
http://www.deri.ie/

DERI TECHNICAL REPORT
DERI TECHNICAL REPORT 2010-12-16, DECEMBER 2010

# ADAPTIVE FRAME OF REFERENCE FOR COMPRESSING INVERTED LISTS

Renaud Delbru[1] Krystian Samp[1]
Stephane Campinas[1] Giovanni Tummarello[1]

**Abstract.** The performance of Information Retrieval systems is a key issue in large web search engines. The use of inverted indexes and compression techniques is partially accountable for the current performance achievement of web search engines. In this paper, we introduce a new class of compression techniques for inverted indexes, the Adaptive Frame of Reference, that provides fast query response time, good compression ratio and also fast indexing time. We compare our approach against a number of state-of-the-art compression techniques for inverted index based on three factors: compression ratio, indexing and query processing performance. We show that significant performance improvements can be achieved.

[1]DERI (Digital Enterprise Research Institute), National University of Ireland, Galway , IDA Business Park, Lower Dangan, Galway, Ireland.

**Acknowledgements**: The work presented in this paper has been funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

Copyright © 2010 by the authors

# 1 Introduction

The performance of Information Retrieval systems is a key issue in large web search engines. The use of appropriate compression techniques is partially accountable for the current performance achievement of web search engines. Compression is not only useful for saving disk space, but it also maximises IO throughput [7] and therefore increases query throughput. In this paper, we introduce a new class of compression techniques for inverted indexes that provides fast query response time, good compression ratio but also fast indexing time.

In the past years, compression techniques have focussed on CPU optimised compression algorithms [9, 1, 18, 3]. It has been shown in [1, 18, 3] that the decompression performance depends on the complexity of the execution flow of the algorithm. Algorithms that require branching conditions tend to be slower than algorithms optimised to avoid branching conditions. In fact, simplicity over complexity in compression algorithms is a key for achieving high performance. The challenge is however to obtain a high compression rate while keeping the execution flow simple.

Previous works [1, 4, 18, 17, 16, 3] have focussed solely on two factors, the compression ratio and the decompression performance, disregarding the compression performance. While decompression performance is essential for query throughput, compression performance is crucial for update throughput. We therefore propose to study compression techniques with an additional third factor, the compression performance. We show that compression performance is also dependent on an optimised execution flow.

## 1.1 Contribution

We study the problem of compression techniques based on three factors: indexing time, compression ratio and query processing time. We compare our approach against a selection of state-of-the-art compression techniques for inverted indexes on two real web datasets. We show that significant performance improvements can be achieved on the indexing time and compression ratio while maintaining one of the fastest query processing time. Our main contributions are the following:

1. We introduce a new class of compression algorithms, the Adaptive Frame of Reference. We show that this compression technique provides the best trade-off between compression speed, decompression speed and compression ratio.

2. We perform a detailed experimental evaluation of a selection of state-of-the-art compression techniques for inverted indexes. We compare their performance based on three factors: compression ratio, indexing performance and query processing performance. The results of our experiment are reported using arithmetic mean as well as standard deviation to quantify the dispersion. ANOVA and pair-wise comparisons are performed to assess if the differences between the algorithms are statistically significant.

# 2 Background

In this section, we first introduce the block-based inverted index that we are using for comparing the compression techniques. We then describe a list of compression techniques, along with their implementation, before introducing the Adaptive Frame Of Reference approach in Sect. 3.

## 2.1 Inverted Index Structure

An inverted index is composed of 1. a lexicon, i.e., a dictionary of terms that allows fast term lookup; and 2. of inverted lists, one inverted list per term. An inverted list is a stream of integers, composed of a list of document identifiers that contain the term, a list of term frequencies in each document and a list of term positions in each document.

In an interleaved index [2], the inverted index is composed of a single inverted file where the inverted lists are stored contiguously. A pointer to the beginning of the inverted list in the inverted file is associated to each term in the lexicon. Therefore, the inverted list of a term can be accessed efficiently by performing a single term lookup on the lexicon. For performance and compression efficiency, it is best to store separately each data stream of an inverted list [2]. In a non-interleaved index organisation, the inverted index is composed of three inverted files, one for each inverted lists (i.e., list of documents, term frequencies and term positions). Each inverted file stores contiguously one type of list, and three pointers are associated to each term in the lexicon, one pointer for each inverted file.

We now describe our implementation[1] of an inverted file on disk. An inverted file is partitioned into blocks, each block containing a fixed number of integers as shown in Fig. 1. Blocks are the basic units for writing data to and fetching data from disk, but also the basic data unit that will be compressed and decompressed. A block starts with a block header. The block header is composed of the length of the block in bytes and additional metadata information that is specific to the compression technique used. Long inverted lists are often stored across multiple blocks, starting somewhere in one block and ending somewhere in another block, while multiple small lists are often stored into a single block. For example, 16 inverted lists of 64 integers can be stored in a block of 1024 integers[2]. We use blocks of 1024 integers in our experiments with respect to the CPU cache, since this was providing the best performance. The overall performance of all the compression techniques were decreasing with smaller block sizes.

## 2.2 Techniques for Inverted List Compression

An inverted index is composed of inverted lists, each one being an ordered list of integers. The main idea of index compression is to encode the list of integers using as few bits as possible. Instead of storing the raw integer in a 32 bit machine word, the goal is to store each integer using the smallest number of bits possible.

[1]This implementation is the one found in the open-source project Apache Lucene.

[2]In that case, the compression techniques are independent of the size of the inverted lists. Therefore, compared to [17], we do not have to rely on secondary compression techniques for short inverted lists.

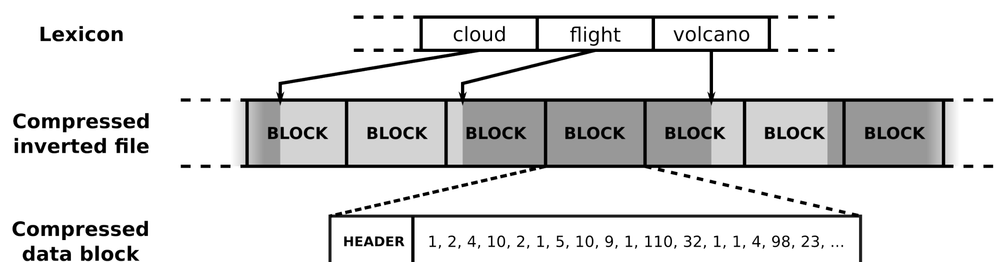



Figure 1: Inverted index structure. Each lexicon entry (term) contains a pointer to the beginning of its inverted list in the compressed inverted file. An inverted file is divided into blocks of equal size, each block containing the same number of values. A compressed block is composed of a block header followed by a compressed list of integers.

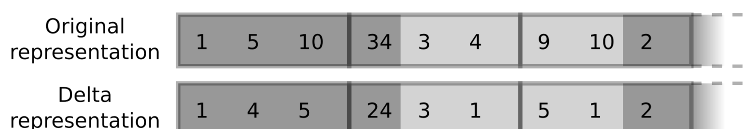



Figure 2: Inverted file with blocks of size 3 using the delta encoding method. The upper inverted file contains the three original lists of integers (1 to 34, 3 to 10, and 2 to ...). The lower inverted file contains the three lists in delta code.

One common practice [12] is to rely on "delta encoding". The delta encoding technique stores the difference between the previous integer and the current one in the list, instead of storing the raw integer. This allows to encode an ordered list of integers using much smaller integers, which theoretically can be encoded in less bits. Delta encoding with block-based inverted files is depicted in Fig. 2. Compression techniques are then used to encode these integers with the smallest number of bits possible. We describe next the five compression algorithms selected for the experiments and discuss their implementation. With respect to the Binary Interpolative Coding [14], it has been shown to provide a very good compression rate and it could have been a reference for comparing compression rates. However, it is very inefficient in decoding, and we found that Rice is competitive enough in term of compression rate to use it as reference.

### 2.2.1 Rice Coding

In Rice, an integer $n$ is encoded in two parts: a quotient $q = \lfloor \frac{n}{2^b} \rfloor$ and a remainder $r = n \bmod 2^b$. The quotient is stored in unary format using $q + 1$ bits while the remainder is store in binary format using $b$ bits. In our implementation, the parameter $b$ is chosen per block such that $2^b$ is close to the average value of the block.

The main advantage of Rice is its very good compression ratio. However, it is in general the slowest method in term of compression and decompression. The main reason is that Rice needs to manipulate the unary word one bit at a time during both compression and decompression, which is very demanding in CPU cycles.

### 2.2.2 Variable Byte Coding (VByte)

Variable Byte compression encodes an integer with a variable number of bytes. VByte is byte-aligned, i.e., coding and decoding is done a byte at a time. Each byte consists of 7 bits to encode the partial binary representation of the integer, and one bit used as status flag to indicate if the following byte is part of the current number. For example, the number $298 = 1 \cdot 2^8 + 42 \cdot 2^0$ is encoded using two bytes $\mathbf{1}0000001$ $\mathbf{0}0101010$. The most significant bit $\mathbf{1}$ of the first byte indicates that the next byte is part of the current integer and must be decoded. The most significant bit $\mathbf{0}$ of the second byte indicates that there is no more bytes to decode for the current integer.

The advantages of VByte are: 1. it is simple to implement; and 2. its overall compression and decompression performance are good. Compared to bitwise techniques like Rice, VByte requires a single branching condition for each byte which is more cost-effective in term of CPU cycles. However, the branching condition leads to branch mispredictions which makes it slower than CPU optimised techniques such as the one presented next. Moreover, VByte has a poor compression ratio since it requires one full byte to encode small integers (i.e., $\forall n < 2^7$).

### 2.2.3 Simple Coding Family

The idea behind the Simple coding is to pack as many integers as possible into one machine word (being 32 or 64 bits). We describe one Simple coding method (referred to as S-64 in our experiments) based on 64-bit machine words, recently introduced in [3]. In our experiments, we report only S-64 results since its overall performance was always superior to Simple9 [1]. In S-64, each word consists of 4 status bits and 60 data bits. The 4 status bits are used to encode one of the 16 possible configurations for the data bits. A description of the 16 configurations can be found in [3]. S-64 wastes generally less bits than Simple9 and therefore provides a better compression ratio.

While providing good compression ratio, decompression is done efficiently by reading one machine word at a time and by using a precomputed lookup table over the status bits in order to select the appropriate optimised routine (one routine per configuration) to decode the data bits using shift and mask operations only.

One disadvantage is that compression cannot be done efficiently. The typical implementation is to use a sliding window over the stream of integers and to find the best configuration, i.e., the one providing the best compression ratio, for the current window. This generally requires repetitive try and error iterations over the possible configurations at each new window. In addition, Simple coding has to perform one table lookup per machine word which costs more CPU cycles than the other techniques presented next.

### 2.2.4 Frame of Reference (FOR)

FOR determines the range of possible values in a block, called a *frame*, and maps each value into this range by storing just enough bits to distinguish between the values [9]. In the case of the delta-encoded list of values, since the probability distribution generated by taking the delta tends

to be naturally monotonically decreasing, one common practice [18, 3] is to choose as frame the range $[0, max]$ where $max$ is the largest number in the group of delta values.[3]

Given a frame $[0, max]$, FOR needs $\lceil \log_2(max + 1) \rceil$ bits, called *bit frame* in the rest of the paper, to encode each integer in a block. The main disadvantage of FOR is that it is sensitive to outliers in the group of values. For example, if a block of 1024 integers contains 1023 integers inferior to 16, and one value superior to 128, then the bit frame will be $\lceil \log_2(128 + 1) \rceil = 8$, wasting 4 bits for each other values.

However, compression and decompression is done very efficiently using highly-optimised routines [18] which avoid branching conditions. Each routine is loop-unrolled to encode or decode $m$ values using shift and mask operations only. Listing 1 and 2 show the routines to encode or decode 8 integers with a bit frame of 3. There is a compression and decompression routines for each bit frame.

Given a block of $n$ integers, FOR determines a frame of reference for the block and encodes the block by small iterations of $m$ integers using the same compression routine at each iteration. Usually, and for questions of performance, $m$ is chosen to be a multiple of 8 so that the routines match byte boundaries. In our implementation, FOR relies on routines to encode and decode 32 values at a time.

The selection of the appropriate routine for a given bit frame is done using a precomputed lookup table. The compression step performs one pass only over the block to determine the bit frame. Then, it selects the routine associated to the bit frame using the lookup table. Finally, the bit frame is stored using one byte in the block header and the compression routine is executed to encode the block. During decompression, FOR reads the bit frame, performs one table lookup to select the decompression routine and executes iteratively the routine over the compressed block.

```
compress3(int[] i, byte[] b)
  b[0] = (i[0] & 7)
       | ((i[1] & 7) << 3)
       | ((i[2] & 3) << 6);
  b[1] = ((i[2] >> 2) & 1)
       | ((i[3] & 7) << 1)
       | ((i[4] & 7) << 4)
       | ((i[5] & 1) << 7);
  b[2] = ((i[5] >> 1) & 3)
       | ((i[6] & 7) << 2)
       | ((i[7] & 7) << 5);
```

Listing 1: Loop-unrolled compression routine that encodes 8 integers using 3 bits each

```
decompress3(byte[] b, int[] i)
  i[0] = (b[0] & 7);
  i[1] = (b[0] >> 3) & 7;
  i[2] = ((b[1] & 1) << 2)
       | (b[0] >> 6);
  i[3] = (b[1] >> 1) & 7;
  i[4] = (b[1] >> 4) & 7;
  i[5] = ((b[2] & 3) << 1)
       | (b[1] >> 7);
  i[6] = (b[2] >> 2) & 7;
  i[7] = (b[2] >> 5) & 7;
```

Listing 2: Loop-unrolled decompression routine that decodes 8 integers represented by 3 bits each

[3]This assumes that a group of values will always contain 0, which is not always the case. However, we found that taking the real range $[min, max]$ was only reducing the index size by 0.007% while increasing the complexity of the algorithm.

### 2.2.5 Patched Frame Of Reference (PFOR)

PFOR [18] is an extension of FOR that is less vulnerable to outliers in the value distribution. PFOR stores outliers as exceptions such that the frame of reference $[0, max]$ is greatly reduce. PFOR first determines the smallest $max$ value such that the best compression ratio is achieved based on an estimated size of the frame and of the exceptions. Compressed blocks are divided in two: one section where the values are stored using FOR, a second section where the exceptions, i.e., all values superior to $max$, are encoded using 8, 16 or 32 bits. The unused slots of the exceptions in the first section are used to store the offset of the next exceptions in order to keep a linked list of exception offsets. In the case where the unused slot is not large enough to store the offset of the next exceptions, a *compulsive exception* [18] is created.

For large blocks, the linked list approach for keeping track of the position of the exceptions is costly when exceptions are sparse since a large number of compulsory exceptions has to be created. [18] proposes to use blocks of 128 integers to minimise the effect. [16] proposes a non-compulsive approach where the exceptions are stored along with their offset in the second block section. We choose the latest approach since it has been shown to provide better performance [16]. During our experimentations, we tried PFOR with frames of 1024, 128 and 32 values. With smaller frames, the compression rate was slightly better than the compression rate of AFOR-1 (which will be presented next). However, the query time performance was decreasing. We therefore decided to use PFOR with frames of 1024 values as it was providing a good reference for query time.

The decompression is performed efficiently in two phases. First, the list of values are decoded using the FOR routines. Then, the list of values is *patched* by: 1. decompressing the exceptions and their offsets and 2. replacing in the list the exception values. However, the compression phase cannot be efficiently implemented. The main reason is that PFOR requires a complex heuristic that require multiple passes over the values of a block in order to find the frame and the set of exceptions providing the highest compression.

# 3 Adaptive Frame of Reference

The Adaptive Frame Of Reference (AFOR) attempts to retain the best of FOR, i.e., a very efficient compression and decompression using highly-optimised routines, while providing a better tolerance against outliers and therefore achieving a higher compression ratio. Compared to PFOR, AFOR does not rely on the encoding of exceptions in the presence of outliers. Instead, AFOR partitions a block into multiple frames of variable length, the partition and the length of the frames being chosen appropriately in order to adapt the encoding to the value distribution.

To elaborate, AFOR works as follow. Given a block $B$ of $n$ integers, AFOR partitions it into $m$ distinct frames and encodes each frame using highly-optimised routines. Each frame is independent from each other, i.e., each one has its own *bit frame*, and each one encodes a variable number of values. This is depicted in Figure 3 by *AFOR-2*. Along with each frame, AFOR encodes the associated bit frame with respect to a given encoder, e.g., a binary encoder. In fact, AFOR encodes (resp., decodes) a block of values by:

1. encoding (resp., decoding) the bit frame;

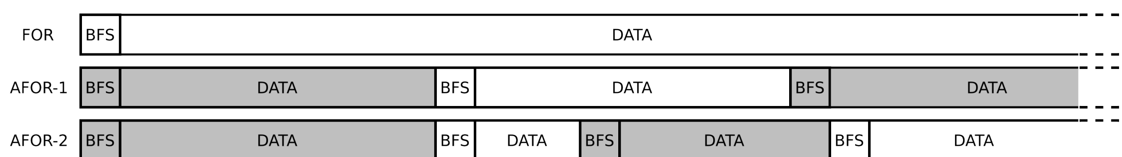



Figure 3: Block compression comparison between FOR and AFOR. We alternate colours to differentiate frames. AFOR-1 denotes a first implementation of AFOR using a fixed frame length. AFOR-2 denotes a second implementation of AFOR using variable frame lengths. *BFS* denotes the byte storing the bit frame selector associated to the next frame, and enables the decoder to select the appropriate routine to decode the following frame.

2. selecting the compression (resp., decompression) routine associated to the bit frame;

3. encoding (resp., decoding) the frame using the selected routine.

Finding the right partitioning, i.e., the optimal configuration of frames and frame lengths per block, is essential for achieving high compression ratio. If a frame is too large, the encoding becomes more sensitive to outliers and wastes bits by using an inappropriate bit frame for all the other integers. On the contrary, if the frames are too small, the encoding wastes too much space due to the overhead of storing a larger number of bit frames. The appropriate strategy is to rely on large frames in the presence of a dense sequence of values, and on small frames in the presence of sparse sequence of values. Also, alternating between large and small frames is not only important for achieving high compression ratio but also for achieving high performance. If frames are too small, the system has to perform more table lookups to select the appropriate routine associated to each frame, and as a consequence the compression and decompression performance decrease. Therefore, it is best to rely on large frames instead of multiple smaller frames when it is possible. To find a block partitioning, our solution uses a local optimisation algorithm which is explained next.

## 3.1 Partitioning Blocks into Variable Frames

Finding the optimal configuration of frames and frame lengths for a block of values is a combinatorial problem. For example, with three different frame lengths (32, 16 and 8) and a block of size 1024, there is $1.18 \times 10^{30}$ possible combinations. While such combinatorial problem can be solved via Dynamic Programming algorithms, the complexity of such algorithms is still polynomial with the size of the block and therefore greatly impacts the compression performance. We remind that we are interested not only by fast decompression speed and high compression ratio, but also by fast compression speed. Therefore, in our experiments, we do not rely on the optimal configuration. Instead, we use a local optimisation algorithm that provides a satisfactory compression rate and that is efficient to compute.

AFOR computes the block partitioning by using a sliding window over a block and determines the optimal configuration of frames and frame lengths for the current window. Given a window of size $w$ and a list of possible frame lengths, we compute beforehand the possible configurations.

For example, for a window size of 32 and three different frame lengths, 32, 16 and 8, there is six configurations: $[32]$, $[16, 16]$, $[16, 8, 8]$, $[8, 16, 8]$, $[8, 8, 16]$, $[8, 8, 8, 8]$. The size of the window as well as the number of possible frame lengths are generally chosen to be small in order to reduce the number of possible configurations. Then, we first compute the bit frames of the smallest frames by doing one pass over the values of the window as shown in the Algorithm 1 (lines 1-5). On the previous example, this means that we compute the bit frames for the configuration $[8, 8, 8, 8]$. The `bitFrames` array stores the bit frame for each of frame of this configuration. Given these bit frames, we are able to compute the bit frames of all the other frames. The second step, lines 6-12 in Algorithm 1, iterates over the possible configurations and estimates the size of each configuration in order to find the optimal one for the current window. Given the previously computed `bitFrames` array, the `EstimateSize` function computes the cost of encoding the window for a given configuration, accounting also the overhead of storing the bit frames. For example, for the configuration $[8, 8, 8, 8]$ with four frames of size 8 each, and with four associated bit frames, $b_1$ to $b_4$, the size of the encoding is computed as follow: $(4 \times 8) + 8 \times \sum_{i=1...4} b_i$, where $8 \times \sum_{i=1...4} b_i$ is the size (in bits) of the four encoded frames and $4 \times 8$ is the overhead (in bits) to store the four bit frames.

This simple algorithm is efficient to compute, in particular if the window size is small and if there is a few number of possible frame lengths. However, it is easy to see that such method does not provide the optimal configuration for a complete block. There is a trade-off between optimal partitioning and complexity of the algorithm. One can possibly use a more complex method for achieving a higher compression if the compression speed is not critical. However, this is not the case for a web search engine where high update throughput is crucial. We decided to use this method since in our experiments we found that a small window size of 32 values and three frame lengths, 32, 16 and 8, were providing satisfactory results in term of compression speed and compression ratio. More details about our implementations of AFOR are given next. As a final reminder, we point out that we are interested by fast compression speed and by presenting the general benefits of AFOR. We therefore do not compare our partitioning scheme with *optimal* partitioning in this paper, and defer this task to an extended version of the paper.

## 3.2 Implementation

We present two different implementations of the AFOR encoder class. We can obtain many variations of AFOR by using various sets of frame lengths and different parameters for the partitioning algorithm. We tried many of them during our experimentation and report here only the ones that are promising and interesting to compare.

### 3.2.1 AFOR-1

The first implementation of AFOR, referred to as AFOR-1 and depicted in Figure 3, is using a single frame length of 32 values. To clarify, this approach is identical to FOR applied on small blocks of 32 integers. This first implementation shows the benefits of using short frames instead of long frames of 1024 values as in our original FOR implementation. In addition, AFOR-1 is used

**Algorithm 1:** The algorithm that finds the best configuration of frames and frame lengths for a window $W$ of size $w$.

```
input  : A window W of size w
input  : The smallest frame length l
output : The best configuration for the window

1  for i ← 0 to w/l do
2      for j ← i × l to (i + 1) × l do
3          bitFrames[i] ← max(bitFrames[i], ⌈log2(W[j] + 1)⌉);
4      end
5  end
6  bestSize ← MaxSize;
7  foreach configuration c of the possible configurations do
8      if EstimateSize(c, bitFrames) < bestSize then
9          bestSize ← EstimateSize(c);
10         bestConf ← c;
11     end
12 end
```

to compare and judge the benefits provided by AFOR-2, the second implementation using variable frame lengths. Considering that, with a fixed frame length, a block is always partitioned in the same manner, AFOR-1 does not rely on the partitioning algorithm presented previously.

#### 3.2.2 AFOR-2

The second implementation, referred to as AFOR-2 and depicted in Figure 3, relies on three frame lengths: 32, 16 and 8. We found that these three frame lengths give the best balance between performance and compression ratio. Additional frame lengths were rarely selected and the performance was decreasing due to the larger number of partitioning configurations to compute. Reducing the number of possible frame lengths was providing slightly better performance but slightly worse compression ratio. There is a trade-off between performance and compression effectiveness when choosing the right set of frame lengths. Our implementation relies on the partitioning algorithm presented earlier, using a window's size of 32 values and six partitioning configurations $[32], [16, 16], [16, 8, 8], [8, 16, 8], [8, 8, 16], [8, 8, 8, 8]$.

#### 3.2.3 Compression and decompression routines

For question of efficiency, our implementations rely on highly-optimised routines such as the ones presented in Listing 1 and 2, where each routine is loop-unrolled to encode or decode a fixed number of values using shift and mask operations only. In our implementation, there is one such routine per bit frame and per frame length. For example, for a bit frame of 3, we need a different set of instructions depending on the size of the frame. For a frame length of 8 values, the routine

encodes 8 values using 3 bits each as shown in Listing 1, while for a frame length of 32, the routine encodes 32 values using 3 bits each. One could use the routines for a frame length of 8 in a loop to encode and decode a frame of 32, instead of using especially made routines for 32 values, but at the cost of 4 loop conditions and 4 function calls.

Since AFOR-1 uses a single frame length, it only needs 32 routines for compression and 32 routines for decompression, i.e., one routine per bit frame (1 to 32). With respect to AFOR-2, since it relies on three different frame lengths, it needs 96 routines for compression and 96 routines for decompression.

#### 3.2.4 Bit frame encoding

We remind that the bit frame is encoded along with the frame, so that, at decompression time, the decoder can read the bit frame and select the appropriate routine to decode the frame. In the case of AFOR-1, the bit frame varies between 1 to 32. However, for AFOR-2, there is 96 cases to be encoded, where cases 1 to 32 refer to the bit frames for a frame length of 8, cases 33 to 63 refer to the bit frames for a frame length of 16, and cases 64 to 96 refer to the bit frames for a frame length of 32. In our implementation, the bit frame is encoded using one byte. While this approach wastes some bits each time a bit frame is stored, more precisely 3 bits for AFOR-1 and 1 bits for AFOR-2, the choice is again for a question of efficiency. Since bit frames and frames are interleaved in the block, storing the bit frame using one full byte enables the frame to be aligned with the start and end of a byte boundary. Another possible implementation to avoid wasting bits is to pack all the bit frames at the end of the block. We tried this approach and report that it provides slightly better compression ratio, but slightly worse performance. Since the interleaved approach was providing better performance, we decided to use this one in our experiment.

#### 3.2.5 Routine selection

A precomputed lookup table is used by the encoder and decoder to quickly select the appropriate routine given a bit frame. Compared to AFOR-1, AFOR-2 has to perform more table lookups for selecting routines since AFOR-2 is likely to rely on small frames of 8 or 16 values when the value distribution is sparse. While these lookups cost additional CPU cycles, we will see in Section 4 that the overhead is minimal compared to the increase in compression ratio provided.

# 4 Experimental Results

This section describes the benchmark experiments which aim to compare the various compression methods described previously. The first experiment measures the indexing performance based on two aspects: 1. the indexing time; and 2. the index size. The second experiment compares the query execution performance. Instead of comparing the raw performances of the compression algorithms, i.e., the number of integers decoded or encoded per millisecond or the average bits per integer, as it is done usually, we decided to compare the performance in real settings, that is when the compression techniques are employed by a web search engine while indexing data and

answering queries. We found that such approach helps to understand better the benefits of each compression technique with respect to a web search engine. In general, the performance depends on the type of the value distribution to encode or decode and on the type of queries to answer. For example, some technique provides good performance on simple value distributions such as the lists of term frequencies, but bad performance on skewed value distributions such as the lists of positions.

**Experimental Settings** The hardware system we use in our experiments is a 2 x Opteron 250 @ 2.4 GHz (2 cores, 1024 KB of cache size each) with 4GB memory and a local SATA disk. The operating system is a 64-bit Linux 2.6.31-20-server. The version of the Java Virtual Machine (JVM) used during our benchmarks is 1.6.0_20. The compression algorithms and the benchmark platform are written in Java and based on the open-source project Apache Lucene.

**Experimental Design** Each measurement is made by 1. flushing the OS cache; 2. initialising a new JVM and 3. warming the JVM by executing a certain number of times the benchmark. The JVM warmup is necessary in order to be sure that the OS and the JVM have reached a steady state of performance (e.g., that the critical portion of code is JIT compiled by the JVM). The implementation of our benchmark platform is based on the technical advices from [5], where more details about the technical aspects can be found.

**Data Collection** We use two real web datasets for our comparison:

**Wikipedia:** set of english language Wikipedia articles ( 2.5 million articles). Its total size is 42GB uncompressed.

**Blog:** set of 44 million blog posts made between August 1st and October 1st, 2008 [6]. Its total size is 142GB uncompressed.

## 4.1 Indexing Performance

The performance of indexing is compared based on the index size (compression ratio), commit time (compression speed) and optimise time (compression and decompression speed). The indexing is performed by adding incrementally 10000 documents at a time and finally by optimising the index. For each batch of documents, the commit operation creates a small inverted index (called an *index segment*). The optimisation merges all the index segments into a single segment. We believe this to be a common operation for incremental inverted index.

We report the results of the indexing experiments in Table 1. The table comprises two columns with respect to the indexing time: the total commit time (*Total*) to add all the documents and the optimisation time (*Opt*). The time collected is the CPU time used by the current thread and comprises the user time and the system time. The index size in Table 1 is studied based on the size of the individual inverted file (document, frequency and position) and on the total index size (by summing the size of the three inverted files). We also provide bar plots to visualise better the differences between the techniques.

| | Wikipedia | | | | | | Blog | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (s) | | Size (Gb) | | | | Time (s) | | Size (Gb) | | | |
| Method | Total | Opt | Doc | Frq | Pos | Total | Total | Opt | Doc | Frq | Pos | Total |
| AFOR-1 | 410 | 114 | 0.353 | 0.109 | 0.698 | 1.160 | 12337 | 4813 | 7.868 | 2.366 | 21.058 | 31.292 |
| AFOR-2 | 409 | 128 | 0.340 | 0.093 | 0.655 | 1.088 | 13040 | 4571 | 7.387 | 2.016 | 19.492 | 28.895 |
| FOR | 443 | 128 | 0.426 | 0.178 | 0.811 | 1.415 | 12741 | 5888 | 9.115 | 3.882 | 26.780 | 39.777 |
| PFOR | 438 | 127 | 0.428 | 0.106 | 0.745 | 1.279 | 13972 | 5387 | 9.097 | 2.464 | 23.975 | 35.536 |
| Rice | 492 | 228 | 0.332 | 0.061 | 0.600 | 0.993 | 14099 | 7127 | 7.145 | 1.546 | 18.160 | 26.851 |
| S9-64 | 462 | 124 | 0.356 | 0.103 | 0.662 | 1.121 | 13152 | 5414 | 7.892 | 2.197 | 19.711 | 29.800 |
| VByte | 433 | 132 | 0.402 | 0.291 | 0.714 | 1.407 | 12320 | 5092 | 8.630 | 5.610 | 22.063 | 36.303 |

Table 1: Total indexing time, optimise time and index size. The best result in each column is underlined using a plain line, while the second best result is underlined using a dashed line.

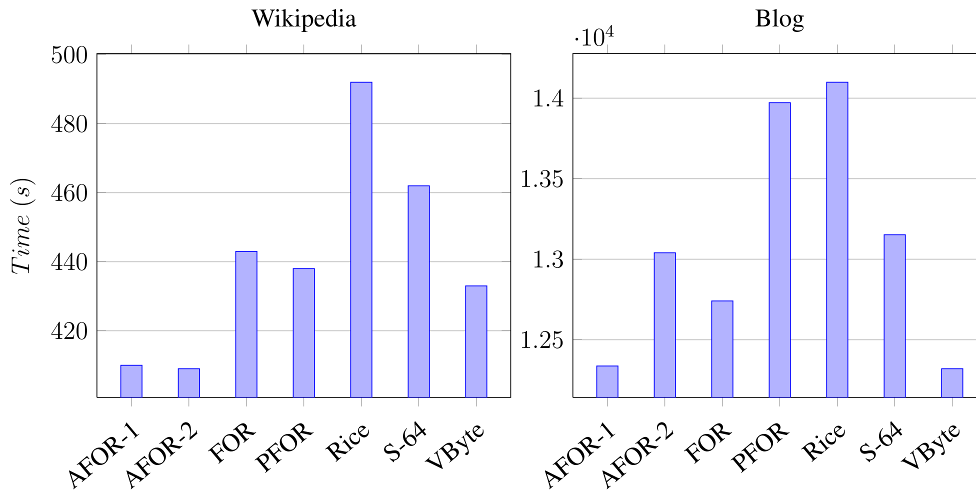



Figure 4: The total time spent to commit batches of 10000 document.

**Commit Time** Figure 4 shows the total time spent by each method. As might be expected, Rice is the slowest method due to its execution flow complexity. It is followed by PFOR then S-64. We can notice the inefficiency of PFOR in term of compression speed. On large datasets (Blog), VByte and AFOR-1 are the best-performing methods while FOR and AFOR-2 provide similar time. On smaller dataset, AFOR-1 and AFOR-2 are the best performing methods while VByte, FOR and PFOR perform similarly. PFOR performs better on small dataset (Wikipedia) due to the smaller number of exceptions to encode.

**Optimisation Time** Figure 5 shows the optimise time for each methods. The time to perform the optimisation step is quite different due to the nature of the operation. The optimisation operation has to read and decompress all the index segments and compress them back into a single segment. Therefore, decompression performance is also an important factor, and algorithms having good decompression speed are more competitive. For example, while PFOR is performing similarly

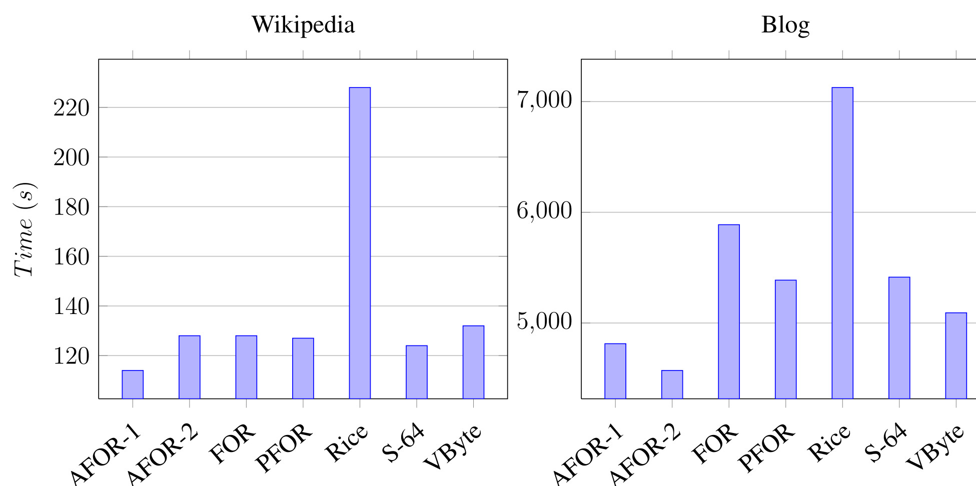



Figure 5: The total time spent to optimise the complete index.

than Rice in term of indexing time on the Blog dataset, it is ahead of Rice and FOR in term of optimisation time. Rice is penalised by its low decompression speed and FOR by its poor compression ratio. Similarly, S-64 provides close or even better performance than VByte due to its faster decompression. VByte is performing well due to its good compression and decompression speed. The best performing methods are AFOR-1 and AFOR-2 which take the advantage due their optimised compression and decompression routines. AFOR-1 is even twice as fast as Rice on the Wikipedia dataset. On large datasets (Blog), AFOR-2 provides better performance than AFOR-1, suggesting that the compression ratio is also an important factor for the optimisation step.

**Compression Ratio** Figure 6 shows the total index size achieved by each method. In term of compression ratio, Rice provides the higher compression on both datasets, and on either the document, frequency or position inverted files as shown in Table 1. Figure 6 shows clearly the problem of FOR against outliers in the value distribution. In Table 1, we can observe that on the document and position inverted files, where the values are likely to be more sparse, FOR compression ratio is higher than any other approaches. However, on the frequency inverted file where the values are fairly small, FOR provides better compression ratio than VByte. We can also observe the benefits of PFOR in comparison with FOR. PFOR is less sensitive to outliers and provides much better compression ratio with a gain of up to 2.8 GB for the position inverted file on Blog and 4.2 GB on the total index size. Compared to PFOR, AFOR-1 provides similar compression ratio on the frequency inverted file, but much better on the document and position files. Compared to AFOR-1, AFOR-2 provides even better compression ratio with a gain of 2.5 GB on the total index size on Blog, and achieves the second best compression ratio after Rice.

**Conclusion** These results show that compression speed is an important factor. Without good compression speed, the update throughput of the index is limited. Also the experiments show that the optimisation operation is dependent of the decompression performance, and its execution time

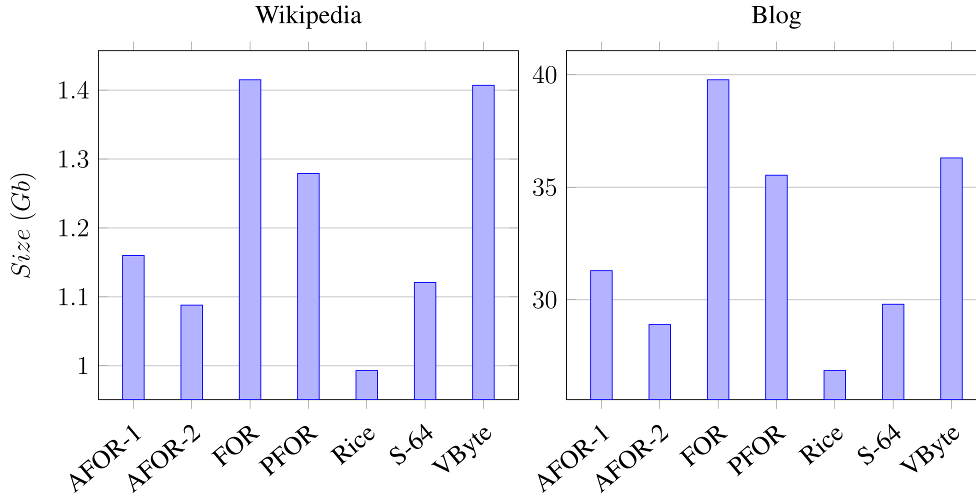



Figure 6: The index size achieved by each compression technique.

can double without a good compression and decompression speed. In addition, the compression ratio must be taken into consideration. While FOR is highly optimised, we can observe that the indexing performance is limited by its poor compression ratio. On large index such as Blog, its low optimisation performance is due to the index size overhead since it has to read and write more data. Overall, the method providing the best balance between indexing time, optimise time and compression ratio is AFOR. AFOR-1 provides fast compression speed and better compression ratio than FOR and PFOR. AFOR-2 provides a notable additional gain in compression ratio and optimise time in exchange of a slight increase of indexing time.

## 4.2 Query Processing Performance

We now compare the decompression performance in real settings, where inverted indexes are answering queries of various complexity. We focus on three types of queries which represent the common queries received by a web search engine: conjunction, disjunction and phrase. Conjunction and disjunction queries help to evaluate the decompression performance on the document and frequency inverted files, and phrase queries on the position inverted file additionally.

**Query Generation** The queries are generated based on word selectivity which determines how many documents match a given keyword. The words are grouped into three selectivity ranges: high, medium and low. We follow the technique described in [8] to obtain the ranges. We first order the words by their descending frequency, and then take the first $k$ words whose cumulative frequency is 90% of all word occurrences as high range. The medium range accounts for the next 5%, and the low range is composed of all the remaining words. For the phrase queries, we follow a similar technique. We first extract all the 2-gram and 3-gram[4] from the data collection. We then

[4] a n-gram is $n$ words that appear contiguously

compute their frequency and sort them by descending frequency. We finally create the three ranges as explained above.

Conjunction and disjunction queries are generated by taking random keywords from the high range group of words. 2-AND and 2-OR (resp. 4-AND and 4-OR) denotes conjunction and disjunction queries with 2 random keywords (resp. 4 random keywords). Similarly, a phrase query is generated by taking random n-grams from the high range group. 2-Phrase (resp. 3-Phrase) denotes phrase queries with 2-gram (resp. 3-gram). Benchmarks involving queries with words from low and medium ranges are not reported here for questions of space, but the performance results are comparable with the ones presented here.

**Query Benchmark Design** For each type of query, we generate a set of 200 random queries which is reused for all the compression methods and perform 100 measurements. Each measurement is made by performing $n$ times the query execution of the 200 random queries, with $n$ chosen so that the runtime is long enough to minimise the time precision error of the OS and machine (which can be 1 to 10 milliseconds) to a maximum of $1\%$. All measurements are made using *warm cache*, i.e., the part of the index read during query processing is fully loaded in memory. The measurement time is the CPU time, i.e., user time and system time, used by the current thread to process the 200 random queries.

Query execution time is sensitive to external events which can affect the final execution time recorded. For instance, background system maintenance or interruptions as well as cache misses or system exceptions can occur and perturb the measurements. All these events are unpredictable and must be treated as noise. Therefore, we need to quantify the accuracy of our measurements. As recommended in [10], we report in Table 2 the arithmetic mean and the standard deviation of the 100 measurements. To assess differences between the algorithms, confidence intervals with $95\%$ degree of confidence can be used. However, in our initial analysis we found that some of these intervals overlap and therefore drawing clear conclusions is prevented [10]. In such a situation, it is recommended to use Analysis of Variance (ANOVA) which is a more robust approach [10]. We use this approach to analyze the data. The design of our query benchmark experiment includes three factors: 1. *Algorithm* having seven levels: AFOR-1, AFOR-2, FOR, PFOR, Rice, S-64, and VByte; 2. *Query* having six levels: 2-AND, 2-OR, 4-AND, 4-OR, 2-Phrase, and 3-Phrase; and 3. *Dataset* having two levels: WIKIPEDIA (small dataset) and BLOG (big dataset). Each condition of the design, e.g., AFOR-1 / 2-AND / WIKIPEDIA, contains 100 separate measurements. We check the assumptions of normality and homogeneity of variances required by ANOVA. The variances between the conditions are not strictly homogeneous (see standard deviations in Table 2), but ANOVA is robust against violations of this assumption [11]. In such a situation it is recommended to log transform measurements before performing ANOVA in order to equalize the variances [13]. We perform additional ANOVA on log transformed measurements. It yields the same results as ANOVA performed on non-transformed measurements.

**Query Benchmark Assessment** There is a significant main effect for *Algorithm* ($F(6, 8316) = 350289$, $p < 0.001$). On average, PFOR is the fastest followed by AFOR-1, FOR, AFOR-2, S-64, VByte, and Rice. There is a significant main effect for *Query* ($F(5, 8316) = 7560023$, $p < 0.001$).

| | 2 - AND | | | 2 - OR | | | 4 - AND | | | 4 - OR | | | 2 - Phrase | | | 3 - Phrase | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Method | $\mu$ | $\sigma$ | MB | $\mu$ | $\sigma$ | MB | $\mu$ | $\sigma$ | MB | $\mu$ | $\sigma$ | MB | $\mu$ | $\sigma$ | MB | $\mu$ | $\sigma$ | MB |
| **Wikipedia** | | | | | | | | | | | | | | | | | | |
| AFOR-1 | 146.3 | 3.1 | 5.7 | 244.1 | 7.5 | 5.8 | 203.3 | 18.0 | 11.3 | 553.9 | 11.6 | 11.4 | 971.6 | 10.3 | 62.6 | 2417.0 | 76.6 | 184.9 |
| AFOR-2 | 153.0 | 11.9 | 5.4 | 262.0 | 6.0 | 5.4 | 212.0 | 3.9 | 10.6 | 558.8 | 3.5 | 10.7 | 970.3 | 5.0 | 58.9 | 2696.0 | 7.2 | 173.3 |
| FOR | 137.1 | 2.0 | 7.7 | 266.6 | 8.0 | 7.7 | 217.3 | 22.6 | 15.1 | 554.7 | 12.4 | 15.2 | 888.1 | 4.1 | 75.3 | 2429.0 | 17.1 | 224.2 |
| PFOR | 138.7 | 12.3 | 6.7 | 265.8 | 3.0 | 6.7 | 199.7 | 3.0 | 13.3 | 549.6 | 9.0 | 13.4 | 908.4 | 5.2 | 66.2 | 2518.0 | 6.1 | 195.2 |
| Rice | 258.1 | 8.0 | 5.0 | 372.5 | 2.6 | 5.0 | 439.9 | 10.4 | 9.8 | 788.0 | 9.2 | 9.9 | 2215.0 | 23.6 | 51.3 | 6234.0 | 9.4 | 149.5 |
| S9-64 | 152.6 | 6.4 | 5.7 | 277.7 | 7.2 | 5.7 | 229.0 | 15.9 | 11.2 | 573.6 | 10.5 | 11.3 | 1009.0 | 41.4 | 60.4 | 2790.0 | 47.8 | 177.1 |
| VByte | 164.7 | 2.0 | 8.6 | 286.8 | 5.6 | 8.7 | 258.6 | 14.4 | 16.8 | 597.3 | 12.0 | 17.0 | 1144.0 | 5.5 | 81.0 | 3113.0 | 77.5 | 240.6 |
| **Blog** | | | | | | | | | | | | | | | | | | |
| AFOR-1 | 195.0 | 2.5 | 12.3 | 461.0 | 8.3 | 13.3 | 276.6 | 21.1 | 21.3 | 1034.0 | 5.6 | 25.4 | 3483.0 | 6.9 | 288.7 | 18934.0 | 309.1 | 1468.8 |
| AFOR-2 | 212.8 | 13.2 | 11.4 | 518.5 | 5.9 | 12.3 | 298.7 | 13.2 | 19.8 | 1057.0 | 6.0 | 23.5 | 3805.0 | 65.5 | 265.5 | 20158.0 | 19.7 | 1334.8 |
| FOR | 199.4 | 9.7 | 15.4 | 502.7 | 8.7 | 16.7 | 290.8 | 29.2 | 26.6 | 1053.0 | 14.7 | 32.0 | 3606.0 | 7.1 | 362.4 | 18907.0 | 264.7 | 1904.4 |
| PFOR | 207.2 | 10.9 | 13.7 | 514.6 | 7.5 | 14.8 | 293.6 | 20.3 | 23.9 | 1033.0 | 5.5 | 28.6 | 3790.0 | 24.4 | 315.9 | 18144.0 | 345.2 | 1657.8 |
| Rice | 433.0 | 11.6 | 10.7 | 725.2 | 16.0 | 11.4 | 622.8 | 4.9 | 18.8 | 1471.0 | 12.7 | 22.1 | 9162.0 | 10.1 | 235.1 | 45689.0 | 30.5 | 1189.8 |
| S9-64 | 225.7 | 14.8 | 12.2 | 530.4 | 4.4 | 13.1 | 313.2 | 10.1 | 21.2 | 1100.0 | 5.5 | 25.1 | 4005.0 | 9.7 | 273.0 | 21260.0 | 314.4 | 1370.6 |
| VByte | 248.7 | 19.4 | 17.1 | 536.4 | 22.1 | 18.6 | 376.4 | 33.4 | 29.1 | 1200.0 | 11.8 | 35.0 | 4400.0 | 7.3 | 355.3 | 21615.0 | 25.9 | 1762.6 |

Table 2: Query execution time in milliseconds per query type, algorithm and dataset. We report for each query type the arithmetic mean ($\mu$), the standard deviation ($\sigma$) and the total amount of data read during query processing (*MB*).

On average, 2-AND query is the fastest followed by 4-AND, 2-OR, 4-OR, 2-Phrase, and 3-Phrase. There is a significant main effect for *Dataset* ($F(1, 8316) = 7250682$, $p < 0.001$). On average, query performance is better on the wikipedia dataset than on the blog dataset. Additionally, interactions between all the factors are significant ($p < 0.001$). We present next the results in more details. For post-hoc pair-wise comparisons, we use Bonferroni adjustment to preserve familywise significance level (p=0.01). Whenever we report that A is faster/slower than B it means that a pair-wise comparison yielded a significant result and thus A is significantly faster/slower than B.

**Query Benchmark Results** We can observe in Table 2 that Rice has the worst performance for every query and dataset. It is followed by VByte. However, Rice performs in many cases twice as slow as VByte. S-64 provides similar performance to VByte on conjunction and disjunction queries but it is significantly faster on phrase queries. However, S-64 stays behind FOR, PFOR and AFOR.

FOR, PFOR and AFOR have relatively similar performances on all the boolean queries and all the datasets. PFOR and AFOR-1 seem to provide generally slightly better performance but the differences are not statistically significant. AFOR-1, PFOR and FOR are faster than AFOR-2 on phrase queries. AFOR-1, PFOR and FOR are relatively similar on phrase queries with FOR and AFOR-1 being faster than PFOR on some queries and the contrary on the other queries. However, AFOR-1 and AFOR-2 are more IO efficient. AFOR-1 reads up to 23% less data than FOR and up to 11% less data than PFOR, and AFOR-2 up to 30% less data than FOR and up to 20% less data than PFOR.

Figure 7 reports the overall query time for each of the compression technique and for each dataset. The overall query time has been computed by summing up the average query time of each

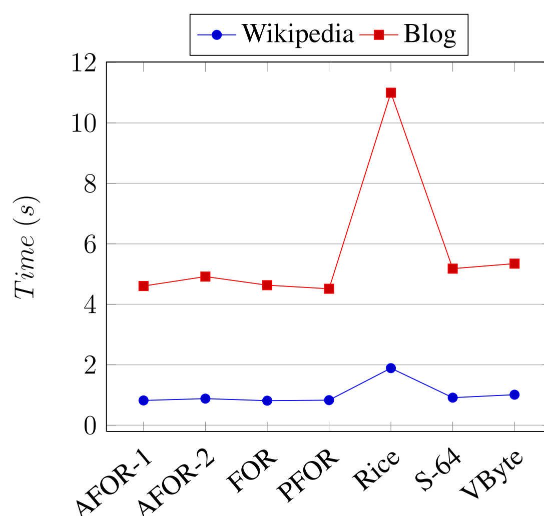


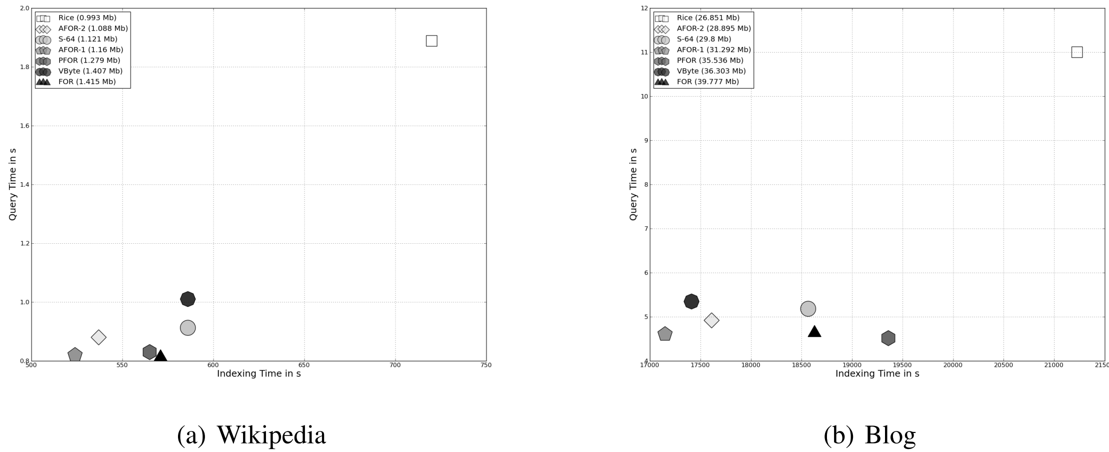

Figure 7: The overall query time achieved by each compression techniques. The overall query time has been obtained by summing up the average query times of each query.

query from Table 2. On this figure, we can distinguish more clearly three classes of algorithms: the techniques based on FOR, a group composed of S-64 and VByte, and finally Rice. The FOR group achieves relatively similar results, with AFOR-2 slightly behind the others.

## 4.3 Performance Trade-Off

We report in Figure 8 the trade-off between indexing time, querying time and compression ratio among all the techniques. The query time has been obtained by summing up the average query times of each query as for Figure 7. The indexing time has been obtained by summing up the commit and optimise time from Table 1. The compression ratio is based on the total size of the index from Table 1. The compression ratio is represented as a gray scale, where white refers to the best compression ratio and black refers to the worst one.

We can distinctively see that AFOR-2 is close to Rice in term of compression ratio, while being one of the fastest in term of indexing and querying time. AFOR-1 provides slightly better indexing and querying time but in exchange of a lower compression ratio. Also, we can observe that VByte offers a good trade-off between compression and decompression time. However, it suffers from its low compression rate. It is interesting to notice that PFOR provides a slightly better querying time and a better compression rate than FOR but at the price of a much slower compression. We can also observe that all the methods, apart from Rice, are clustered on the query time dimension and more sparse on the indexing time dimension, suggesting a limited difference in term of decompression performance and a greater variance in term of compression performance among the algorithms. To conclude, AFOR-2 seems to offer the best compromise between querying time, indexing time, and compression rate.




(a) Wikipedia (b) Blog

Figure 8: A graphical comparison of the compression techniques showing the trade-off between indexing time, querying time and compression ratio. The compression ratio is represented over a gray scale, black being the worst compression ratio and white being the best.

# 5 Discussion and Future Work

While AFOR provides better compression ratio than PFOR, AFOR-2 is slower than PFOR on phrase queries and therefore on position inverted files, suggesting that PFOR exception management is slightly more efficient than the variable frame length approach of AFOR-2 on very sparse lists. The number of table lookups in AFOR-2 costs more than the decoding and patching of the exceptions in PFOR. Future work will investigate how to combine the two approaches in order to keep the best of the two methods.

In general, even if FOR has more data to read and decompress, it still provides the best query execution time. The reason is that our experiments are performed using warm cache. We therefore ignore the cost of disk IO accesses and measure exclusively the raw decompression performance of the methods. With a cold cache, i.e., when IO disk accesses have to be performed, we expect a drop of performance for algorithms with a low compression ratio such as FOR and PFOR compared to AFOR-2. Future work will investigate this aspect.

Compression and decompression performance do not only depend on the compression ratio, but also on the execution flow of the algorithm and the number of cycles needed to compress or decompress an integer. Therefore, CPU-optimised algorithms providing good compression ratio increase the update and query throughputs of web search engines. In that context, AFOR seems to be a good candidate since it is well balanced in all aspects: it provides very good indexing and querying performance and one of the best compression ratio.

The Simple encoding family is somehow similar to AFOR. At each iteration, S-64 encodes or decodes a variable number of integers using CPU optimised routines. AFOR is however not tied to the size of a machine word, is simpler to implement and provides better compression ratio, compression speed and decompression speed.

Another interesting property of AFOR which is not discussed in this paper is its ability to skip quickly over chunks of data without having to decode them. This is not possible with techniques such as Rice, VByte or PFOR. AFOR has to decode the bit frame to know the length of the following frame, and is therefore able to deduce the position of the next bit frame. Such characteristic could be leveraged to simplify the self-indexing of inverted files [15]. In addition, one of the challenge with the self-indexing technique is how to place synchronisation points, called skips, over the list of integers. We would have to study the impact of aligning the synchronisation points with the partitioning produced by AFOR-2.

# 6 Conclusion

We presented AFOR, a novel class of compression techniques for inverted lists. AFOR is specifically designed to increase update and query throughput of web search engines. We compare AFOR to alternative approaches, and show experimental evidences that AFOR provides well balanced performance over three factors: indexing time, querying time and compression ratio. In addition, AFOR is simple to implement and could become a new compression method of reference.

With respect to FOR, it is known that many little variations have been in use and that some of them might be similar to AFOR-1 or AFOR-2. However, none of them have been documented. In this paper we propose an improvement based on an adaptive approach to FOR and provide an extensive experimental coverage.

AFOR is a straightforward extension of FOR, and we admit that the technique is rather simple. However, we stress that this was the design requirement for our extension since we believe that simplicity over complexity is crucial for achieving high performance in compression algorithms. The "adaptive" part of AFOR does not change the class of computational complexity of FOR during compression as compared to PFOR, while providing higher compression ratio. The source code of the benchmark platform and of the compression algorithms as well as the raw experimental results are available on demand.

# References

[1] Vo Ngoc Anh and Alistair Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151–166, January 2005.

[2] Vo Ngoc Anh and Alistair Moffat. Structured index organizations for high-throughput text querying. 4209:304–315, 2006.

[3] Vo Ngoc Anh and Alistair Moffat. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, 2010.

[4] Paolo Boldi and Sebastiano Vigna. Compressed Perfect Embedded Skip Lists for Quick Inverted-Index Lookups. In Mariano Consens and Gonzalo Navarro, editors, *Proceedings of the 12th International Conference on String Processing and Information Retrieval*, volume 3772 of *Lecture Notes in Computer Science*, pages 25–28, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Brent Boyer. Robust Java benchmarking, 2008.

[6] Kevin Burton, Akshay Java, and Ian Soboroff. The ICWSM 2009 Spinn3r Dataset. In *Third Annual Conference on Weblogs and Social Media (ICWSM 2009)*, San Jose, CA, May 2009. AAAI. http://icwsm.org/2009/data/.

[7] Stefan Büttcher and Charles L. A. Clarke. Index compression is good, especially for random access. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management - CIKM '07*, pages 761–770, New York, New York, USA, 2007. ACM Press.

[8] Vuk Ercegovac, David J. DeWitt, and Raghu Ramakrishnan. The TEXTURE benchmark: measuring performance of text queries on a relational DBMS. In *Proceedings of the 31st international conference on Very large data bases*, pages 313–324. VLDB Endowment, 2005.

[9] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. Compressing relations and indexes. In *Proceedings of the 14th International Conference on Data Engineering*, pages 370–379, Washington, DC, USA, 1998. IEEE Computer Society.

[10] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[11] Harold R. Lindman. *Analysis of variance in complex experimental designs*. W.H. Freeman, 1974.

[12] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[13] Scott E. Maxwell and Harold D. Delaney. *Designing Experiments and Analyzing Data: A Model Comparison Perspective*. 2004.

[14] Alistair Moffat and Lang Stuiver. Binary Interpolative Coding for Effective Index Compression. *Information Retrieval*, 3(1):25–47, 2000.

[15] Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4):349–379, 1996.

[16] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th international conference on World Wide Web - WWW '09*, pages 401–410, New York, New York, USA, 2009. ACM Press.

[17] Jiangong Zhang, Xiaohui Long, and Torsten Suel. Performance of compressed inverted list caching in search engines. In *Proceeding of the 17th international conference on World Wide Web - WWW '08*, pages 387–396, New York, New York, USA, 2008. ACM Press.

[18] M. Zukowski, S. Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59, Washington, DC, USA, 2006. IEEE Computer Society.