A General SIMD-based Framework for Accelerating Compression Algorithms

WAYNE XIN ZHAO, Renmin University of China
XUDONG ZHANG, Yahoo! China
DANIEL LEMIRE, Université du Québec
DONGDONG SHAN, Alibaba Group
JIAN-YUN NIE, Université de Montréal
HONGFEI YAN, Peking University
JI-RONG WEN, Renmin University of China

Compression algorithms are important for data oriented tasks, especially in the era of “Big Data”. Modern processors equipped with the powerful SIMD instruction sets, provide us an opportunity for achieving better compression performance. SIMD-based optimization on compression algorithms has been explored in some studies to some extent. Following these pioneering studies, we propose a general compression framework to improve instruction-level parallelizability of compression algorithms. By instantiating the framework, we have developed several novel compression algorithms, called Group-Simple, Group-Scheme, Group-AFOR, and Group-PFD respectively, and implemented their corresponding vectorized versions. We evaluate the proposed algorithms on two public TREC datasets, a Wikipedia dataset and a Twitter dataset. With competitive compression ratios and encoding speeds, our SIMD-based algorithms outperform state-of-the-art non-vectorized algorithms with respect to decoding speeds.

Categories and Subject Descriptors: **E.4** [Coding and Information Theory]: Data Compaction and Compression; **H.3.1** [Information Storage and Retrieval]: Content Analysis and Indexing| indexing methods; **C.1.2** [Processor Architectures]: [Single-instruction stream, multiple-data-stream processors (SIMD)]

General Terms: Algorithms, Performance, Measurement, Experimentation

Additional Key Words and Phrases: SIMD, inverted index, index compression, integer encoding

1. INTRODUCTION

In recent years, we have witnessed an explosive growth of Web data. The overwhelming data raises compelling computational challenges to Web search engines. Although nowadays CPUs have very powerful computational ability, the performance of Web search engines is largely inhibited by slow disk accesses, and the bandwidth of data transferred from disk to main memory becomes the limiting factor for the efficiency.

For search engines, researchers mainly focus on the primary structure, i.e. the inverted index. Various techniques have been shown to be effective to improve the performance of inverted index, especially index compression [Navarro et al. 2000]. Compression algorithms can reduce the space of posting lists, and therefore reduce the transfer of data from disk to memory [Manning et al. 2008, p. 85; Zhang et al. 2008]. To improve the efficiency of query evaluation, many studies have been devoted to developing efficient index compression algorithms [Dean 2009; Navarro 2000; Anh and Moffat 2005; Stepanov et al. 2011]. In particular, recent hardware designs offer hardware-level parallelism. For example, the SSE instruction sets [Intel 2010] in

Author’s addresses: W.X Zhao (Contact Author), Renmin University of China, email: batmanfly@gmail.com; X. Zhang,Yahoo! China, Beijing; D. Lemire, Université du Québec; D. Shan, Alibaba Group; Jian-Yun Nie, Université de Montréal; H. Yan, Peking University; Ji-Rong Wen, Renmin University of China. Xin Zhao and Xudong Zhang contributed equally to this work and should be considered as co-first authors.

Intel's processors is a Single Instruction Multiple Data (SIMD) collection of instructions, which has accelerated 3D computing [Ma et al. 2002], audio and video processing [Liu et al. 2006], and other CPU-intensive tasks [Chatterjee et al. 2005]. SSE instruction sets operate on 128-bit registers, which are able to process four 32-bit integers simultaneously. Inspired by this observation, some pioneering studies have incorporated SIMD-based optimization into compression algorithms [Stepanov et al. 2011; Schlegel et al. 2010]. These studies indicate that index compression can benefit from vectorization. We aim to develop a compression framework to leverage SIMD instructions present in modern processors.

To design a suitable storage layout and following earlier work, we store separately control patterns from compressed data and adopt a k-way vertical data organization [Lemire and Boystov 2013, Schlegel et al. 2010], which makes algorithms easily parallelizable by SIMD instructions. Based on such a storage layout, we present a detailed description of the framework and describe strategies to best leverage SIMD-based parallelism. We have instantiated the framework with algorithms of four categories, which covers most of the important compression algorithms in practice. First, we develop two novel compression algorithms (or algorithm families), i.e. Group-Simple and Group-Scheme. These two algorithms can be directly optimized by SIMD instructions. Group-Simple is extended from the traditional Simple algorithms [Anh and Moffat 2005; Anh and Moffat 2006], which can be considered as a word-aligned algorithm; Group-Scheme originates the idea of Elias Gamma [Elias 1975], which can be considered as a family and contains both bit-aligned and byte-aligned variants. Group-Scheme is flexible enough to adapt to different data sets by adjusting two control factors, i.e., compression granularity and length descriptors. We further present the SIMD-based implementations of Group-Simple and Group-Scheme respectively denoted as SIMD-Group-Simple and SIMD-Group-Scheme. Besides the above two kinds of algorithms, we also develop the group based versions and corresponding vectorized versions of AFOR [Delbru et al. 2012] and PForDelta [Zukowski 2006] in the general framework. To evaluate the proposed methods, we construct extensive experiments on four public datasets.

The contribution of this paper is summarized as follows:

— Our framework provides a general way to vectorize traditional compression algorithms.

— We develop several novel compression algorithms based on the general compression framework, namely Group-Simple, Group-Scheme, Group-AFOR and Group-PFD. These algorithms cover four major categories of traditional compression algorithms.

— We proposed and implemented the corresponding vectorized versions of the proposed algorithms, i.e. SIMD-Group-Simple, SIMD-Group-Scheme, SIMD-Group-AFOR and SIMD-Group-PFD. We also examine several important implementation ideas for optimizing the SIMD based algorithms. To the best of our knowledge, it is the first study which has implemented such a comprehensive coverage of vectorized compression algorithms in a unified framework.

— We construct extensive experiments on four very diverse datasets, including TREC standard data sets GOV2 and ClueWeb09B, Wikipedia dataset and Twitter dataset. Experiments have shown that our novel SIMD-based algorithms achieve fast decoding speed, competitive encoding speed and compression ratio compared with several strong baselines.

The remainder of the paper is organized as follows. We review the technical background and related studies in Section 2. We present the general compression framework in Section 3. Section 4, Section 5 and Section 6 present the proposed algorithms include Group-Simple, Group-Scheme, Group-AFOR and Group-PFD. We carefully study the encoding format and the compression/decompression procedure together with their corresponding SIMD-based versions. Section 7 presents the experimental results and detailed analysis. Section 8 concludes the paper and discusses possible extensions.

2. RELATED WORK

Inverted indexes provide a general way to index various types of documents, whether an article in Wikipedia, a tweet on Twitter, a news article in the New York Times, a status update in Facebook, etc. Although these types of text documents are different in content coverage and presentation format, we can represent all of them as token sequences and create an *inverted index* to facilitate their access. We first introduce the index representations for documents, and then review the existing compression algorithms in four categories.

2.1 Index Representations for Web documents

Inverted indexes, as the primary structure of search engines, include two important components: a dictionary and a collection of posting lists. A term in the dictionary corresponds to a unique posting list. A posting list is composed of a sequence of postings, each posting might correspond to a triplet: <$DocID$, TF_t, [pos_1, pos_2, …, pos_{TF}]>. $DocID$ is the document identifier, TF_t is the document frequency of term, and pos_i denotes the document position of the ith occurrence of term t. We mainly focus on the compression of the frequency and position numbers in the posting lists. An integer can be encoded in either binary or unary, and the *effective bit length* (also known as *effective bit width*) of an integer is the minimum number of binary bits needed to represent it.

The postings in a posting list are usually sorted in an ascendant order of the value of the *DocID*s. A commonly adopted technique is to perform *d-gap* on posting lists [Manning et al. 2008, p. 96]. Given a strictly increasing positive integer sequence d_1, d_2, …, d_n, the d-gap operation replaces each integer d_i with d_i - d_{i-1}, where $i > 1$. With such a representation, our task is to design ways *to effectively and efficiently compress postings lists in inverted index*. The problem is of general interest: similar techniques are applicable to other fields, such as database compression [Lemke et al. 2011, Raman et al. 2013].

Typically, there are two types of data layout for storing a sequence of encoded integers. *Horizontal layout* stores n encoded integers according to the original order of the integer sequence, while *k-way vertical layout* distributes n consecutive integers to k different groups. In Figure 1, we present an illustrative example of a 4-way vertical layout. Int1 ~ Int8 denote 8 encoded integers and each four consecutive integers are stored in four different groups respectively.

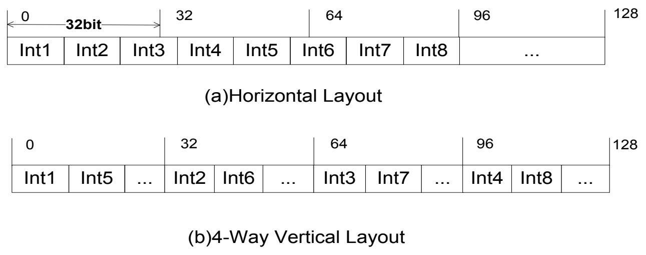


Fig. 1. Horizontal and vertical data layout.

(Will be updated later)

To evaluate the performance of compression algorithms, there are three widely used metrics, namely decoding speed, encoding speed and compression ratio. Decoding/encoding speed measures the processing rate of integers by an algorithm. The compression ratio is defined as the ratio between the uncompressed size and compressed size, and usually expressed as the average number of bits per encoded integer [Manning et al. 2008, p. 87].

2.2 Bit-aligned codes

In bit-aligned codes, the *bit* is the minimum unit to represent an integer. Such work can be traced back to Golomb coding [Witten et al. 1999, p. 121] and Rice coding [Rice and Plaunt 1971]. Golomb coding encodes an integer by two parts, i.e. the quotient and the remainder. The quotient is unary encoded, and the remainder is binary encoded. The bit length depends on the divisor M, which is commonly set to 0.69 times the average value of all integers. Rice coding [Rice and Plaunt 1971] further requires M to be a power of two to acceleration the computation. Golomb coding and Rice coding have high compression ratio, but their compression/decompression speed is slow.

Elias Gamma [Elias 1975] encodes an original integer x with two parts. The first part is the unary coding of the effective bit length, and the second part is the natural binary representation of x without the leading 1.

Schlegel et al. [Schlegel et al. 2010] proposed a vectorized version of Elias Gamma coding, called k-Gamma. k-Gamma encodes a group of k successive integers at a time. It first calculates the effective bit length b of the maximum integer in the group, and represents each integer with b bits. Then, the value of b is encoded in unary and the low b bits of each of k integers are encoded in binary. Schlegel et al. adopted the vertical data layout to keep k integers word-aligned, and applied SIMD instructions to vectorize for storing and loading the k integers. The performance bottleneck of k-Gamma lies in the loading and storing of the unary bit length b. As we will show later, k-Gamma can be viewed as one special variant of our proposed Group-Scheme, and we optimize Group-Scheme with the packed decoding method for length descriptors.

2.3 Byte-aligned codes

Byte-aligned codes represent an integer in bytes. Variable Byte (VB) encoding [Manning et al. 2008, p. 96] uses bytes to represent a non-negative integer, and the most significant bit of a byte is the continuation bit to indicate whether it is the last byte while the remaining bits store the natural binary representation of the integer.

Group Variable Byte (GVB) [Dean 2009] aggregates the flag bits of a group of integers into a byte called *control byte*. When compressing 32-bit integers, a control byte consists of four 2-bit descriptors, where each descriptor represents the number of bytes needed for an original integer in binary notation (00 for 1 byte, 01, for two bytes, 10 for three bytes and 11 for four bytes). GVB encodes and decodes groups of four integers.

If the descriptor are unary coded, they occupy 1 to 4 bits instead of 2 bits, and the number of integers in a group varies from 2 to 8. This variant is called GVB-Unary [Scholer et al. 2002]. GVB-Unary includes two variants: G8IU and G8CU.

Stepanov et al. exploited SIMD instructions to accelerate the decoding speed of the above three variants of GVB [Stepanov et al. 2011]. On x64 processors, integers packed with GVB can be efficiently decoded using the SSSE3 shuffle instructions: *pshufb*. We call the implementations with vectorized decoding as SIMD-GVB-Binary (or SIMD-GVB), SIMD-G8IU and SIMD-G8CU respectively in this paper.

2.4 **Word-aligned codes**

In word-aligned codes, we try to encode as many integers as possible into a 32-bit or 64-bit word. Simple-9 [Anh and Moffat 2005; Anh and Moffat 2006] divides a 32-bit codeword into two parts: a 4-bit selector/pattern segment and a 28-bit data segment. It sets up 9 different selectors to instruct the encoding of consecutive integers in the 28-bit data segment. Zhang et al. proposed Simple-16 [Zhang et al. 2008] improving Simple-9 by extending the number of selectors from 9 to 16.

Anh et al. [Anh and Moffat 2010] used a 64-bit word (a.k.a. Simple-8b and they refer to 32-bit Simple family as Simple-4b). A codeword consists of a 4-bit selector segment and a 60-bit data segment.

Previous studies showed that the Simple family has good overall performance w.r.t compression/decompression speed and compression ratio.

2.5 **Frame fixed bit-length codes**

A frame refers to a group of integers with the same effective bit length. This category includes PackedBinary [Anh and Moffat 2010], PForDelta [Heman 2005; Zukowski 2006; Zhang et al. 2008; Yan et al. 2009], VSEncoding [Silvestri and Venturini 2010] and AFOR [Delbru et al. 2012]. PackedBinary encodes a group of integers using the same effective bit length of *b* bits. However, PackedBinary cannot compress well when there are infrequent large integers (called exceptional values).

To deal with exceptional values, Zukowski et al. [Zukowski 2006] proposed PFORDelta (PFD), which separates normal integers from exceptional integers. The normal integers are still encoded with the same bit length, but the exceptional integers are kept in a global exception array and processed separately. Zukowski's implementation does not compress the exceptional values. As a follow-up, Zhang et al. [Zhang et al. 2008] use 8, 16 and 32 bits to store exceptions according to the maximum exceptional values. Yan et al. [Yan et al. 2009] proposed two new variants called NewPFD and OptPFD. They use the same bit length for a block of 128 integers rather than for all integers. The difference between NewPFD and OptPFD lies in the selection of bit length *b*. NewPFD determines *b* by requiring more than 90% of the integers can be held in *b* bits, while OptPFD determines *b* by optimizing the overall compression ratio.

Lemire et al. [Lemire and Boystov 2013] used SIMD instructions to optimize Packed Binary and PForDelta discussed above. They proposed a novel vectorized algorithm called SIMD-BP128 for fast decompression. They aggregate 128 successive integers as a frame and use vertical layout (in Section 4.1) to pack them with a unified bit length b for the frame. For better compression ratios, they further proposed another new vectorized variant called SIMD-FastPFor, in which they design effective techniques to store the exceptional values.

Packed Binary and PFORDelta adopt a fixed block size (i.e. the number of integers in a block) in contrast to approaches using varying block sizes to improve compression ratio. Fabrizio Silvestri et al. proposed VSEncoding [Silvestri and Venturini 2010], which uses a dynamic programming approach to partition a list of integers into blocks. Block lengths are chosen from the set {1, 2, 4, 8, 12, 16, 32, 64}. Similarly, Delbru et al. proposed AFOR (Adaptive Frame of Reference) [Delbru et al. 2012], which uses only three different block sizes: {8, 16, 32}.

3. A GENERAL SIMD-BASED COMPRESSION FRAMEWORK

In this section, we present a general compression framework, which is designed to incorporate SIMD-based vectorization and highly motivated by the pioneering studies on SIMD-based compression algorithms [Schlegel et al. 2010, Stepanov et al. 2011, Lemire and Boystov 2013]. We borrow and generalize the core ideas of previous SIMD-based algorithms, and propose a general framework to vectorize compression algorithms by utilizing SIMD instructions. We first describe the encoding format, the encoding procedure and the decoding procedure of this framework. Then we discuss how to leverage SIMD instructions for vectorization.

3.1 Encoding Format and Layout Transformation

The storage layout of a sequential algorithm usually consists of control sections and data sections. Data sections store the encoded integers in natural binary data, while control sections (also called *pattern section*) code the information for the start position and end position of each encoded integer in data sections. Typically, many compression algorithms such as VB encoding [Scholer et al. 2002] and Rice encoding [Rice and Plaunt 1971] store a control section and a data section as a whole. As previous studies have shown, it is relatively easier to incorporate SIMD-based vectorization by separating control sections from data sections [Schlegel et al. 2010]. We follow the similar idea and further adopt the interleaving storage technique. Next we introduce how to modify the sequential layout in order to apply SIMD-based vectorization.

Let S denote a sequential algorithm. First, we assign several consecutive data sections to a group and enlarge each data section to the maximum section size in the group. Second, we store all the data sections consecutively in *data area* and store the all control sections consecutively in *control area*. Figure 2 presents an illustrative example. Assume that we have four compressed data sections with m_1, m_2, m_3 and m_4 bits respectively (see Figure 2(a)). Let m is the maximum section size, i.e. m=max(m_1,m_2,m_3,m_4). We enlarge each of these four data sections to m bits, and modify the control section accordingly (see Figure 2(b)). Then we separate data sections from control sections (see Figure 2(c)). The data area can be viewed as a sequence of consecutive 128-bit words. Finally, we modify the storage layout of the data area (shown in Figure 2(d)):

— The data area is divided into consecutive 128-bit segments.
— A segment is further split into four 32-bit data sub-segments. The four sub-segments will share the same compression pattern.
— The data area adopts 4-way vertical storage layout.

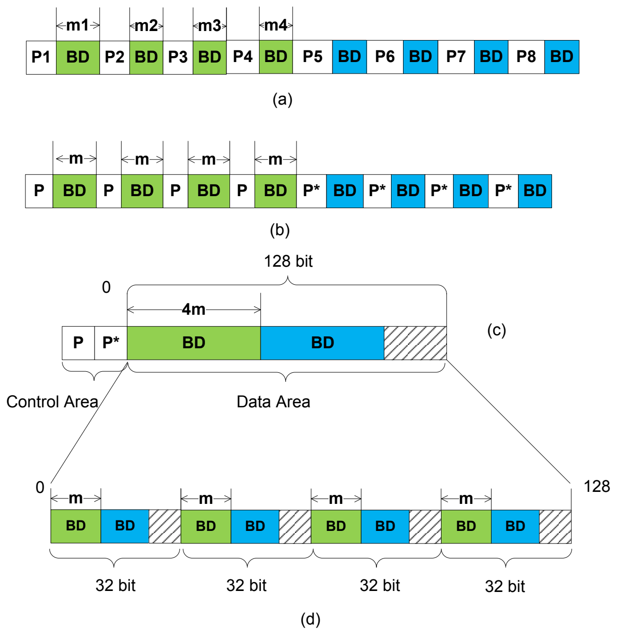

m1
m2
m3
m4
P1 BD P2 BD P3 BD P4 BD P5 BD P6 BD P7 BD P8 BD
(a)
m
P BD P BD P BD P BD P* BD P* BD P* BD P* BD
(b)
128 bit
0
4m
P P* BD BD
(c)
Control Area
Data Area
0
128
BD BD
32 bit
(d)

Fig. 2. An illustrative example for storage layout format of the general compression framework. *P* denotes a pattern section and *BD* denotes a binary data section.

3.2 Encoding Procedure

The encoding procedure of the general compression framework is described as follows:

[Step 1] Find the maximum number of each four integers (called *quad max*) from the original integer array, and build the *quad max array* to hold these numbers, denoted by *MaxArr*. We use the 4-way vertical layout and consider four integers as a group.
[Step 2] Run the sequential compression algorithm *S* on *MaxArr* and derive the corresponding compression patterns for each group. Since each four integers in a group share the same pattern (i.e. the same effective bit length), we can process the *MaxArr* for pattern selection instead of the original array. The generated patterns are stored in the pattern area.
[Step 3] With the generated compression patterns, run *S* to encode each integer in the original array by using the corresponding pattern. We interleave the *k*th, *k+1*th, *k+2*th and *k+3*th integers of a group into four consecutive 32-bit words where *k*=1, 5, 9, 13 ..., etc. *This step can be vectorized by SIMD instructions in the vectorized algorithm.*

3.3 Decoding Procedure

The decoding procedure is described as follows:

[Step 1] Load and obtain the compression patterns for each four 32-bit sub-segments in data area.
[Step 2] For each four 32-bit sub-segments in data area, run *S* to simultaneously decode a four-multiple number of integers. *This step can be potentially vectorized by SIMD instructions in the vectorized algorithm.*
[Step 3] (Optional) According to the original algorithm *S*, we perform specific operation on exceptional bits (e.g. *PForDelta*).

3.4 SIMD Encoding and Decoding

Based on the above storage layout, we study how to apply SIMD instructions for vecotrization. SIMD (Single Instruction, Multiple Data) instructions are widely supported by modern processors. In particular, our SIMD-based algorithms focus on the SSE [Intel 2010] on Intel-compatible processors. Instructions in SSE operate on 128-bit registers (called *XMM registers*), which make it possible to process four 32-bit integers simultaneously. This is the key point for vectorized compression algorithms. We present the SSE instructions used in our algorithms as follows:

- **MOVDQA *dst src***: Copy from 128-bit data source *src* to 128-bit *dst*. *src* and *dst* must be 16-byte aligned, and cannot be memory address at the same time (it requires at least one register).
- **MOVDQU *dst src*:** Same as MOVDQA except that *src* and *dst* are allowed to be 16-byte unaligned.
- **PSRLDQ/PSLLDQ *xmm1 imm8/xmm2***: Regard *xmm1* as an array of four 32-bit integers, and logically shift each integer right/left according the value of immediate *imm8* or *xmm2* register.
- **PAND/POR *xmm1 xmm2***: execute AND/OR operation on the two 128-bit XMM registers.

Consider Step 3 of the encoding procedure (See Section 4.3). Without SIMD instructions, we have to loop four times to encode each four integers into four 32-bit sub-segments respectively. It has been noted that a 128-bit data segment can be loaded into 128-bit XMM register, which is particularly useful for the vectorization of the sequential compression algorithms [Schlegel et al. 2010]. By adopting the 4-way vertical layout, we can process four 32-bit sub-segments simultaneously: we are able to vectorize the shift and mask operations for each four integers with SIMD instructions, which yields a 75% reduction in the number of instruction operations. The 128-bit data segment will be kept in the XMM register and not be written back to memory until all the integers in it are encoded, which further reduces the number of memory addressings in the destination array. Similarly, in Step 2 of the decoding procedure, we can apply SIMD instructions to vectorize the shift and mask operations for decoding each four integers (See Section 4.4), which also yields a 75% reduction in the number of instruction operations. With the 4-way vertical layout, the use of the 128-bit XMM registers facilitates the vectorized processing. In our implementation we can further optimize the vectorized encoding and decoding by reducing the operations on the XMM registers.

3.5 Discussion and Extension

The above compression framework provides a general way to vectorize sequential compression algorithms. The major efforts to apply this framework are to adapt a sequential algorithm to the above storage layout.

Note that our discussions focus on 32-bite integers. When integers are represented in 8/16/64 bits (corresponding to the data types of *char*, *short* and *long long* respectively in *C/C*++), the framework is still applicable by simply modifying the number of sub-segments to 16/8/2 and bit-width of sub-segments to 8/16/64 bits respectively. The size of a data segment is set to 128 bits to match the 128-bit XMM register supported by the SSE instructions. When used in different computer architectures (e.g. non-Intel architecture like ARM and PowerPC), the data segment size might need to be changed accordingly and the same for the number of ways in vertical storage layout.

3.6 Overall organization of the following sections

In Table I, we categorize commonly used (non-SIMD) compression algorithms into four categories. We instantiate the proposed compression framework on several sequential compression algorithms from these four categories. The roadmap of the following sections is listed as follows:

– *Word-aligned*: In Section 5, we propose the *Group-Simple* algorithm, which extends the Simple-9 algorithm.

– *Bit/Byte-aligned*: In Section 6, we propose the *Group-Scheme* family, which originates from the ideas of Elias Gamma and Group Variable Byte algorithms.

– *Frame fixed-bit width*: In Section 7, we propose the *Group-AFOR* and *Group-PFD* based on AFOR and PForDelta respectively.

Table I. Algorithm categorization with the corresponding instantiations in our framework. We mark the wrapped algorithms in bold and present specific modification points to fit into the framework.

Algorithm category	**Sequential algorithms**	**Instantiations in our framework**	**Specific modification**
Bit-aligned	Golomb Rice **Elias Gamma** k-Gamma	Group-Scheme	● Incorporate different compression granularities and length descriptors into the encoding format.
Byte-aligned	Variable Byte **Group Variable Byte**		
Word-aligned	**Simple-9** Simple-16 Simple-8b	Group-Simple	● Provide ten optional controlling patterns and the effective bit length can be up to 32-bits (Simple only supports a maximum bit length of 28 bits).
Frame Fixed-bit-width	**PackedBinary** **PForDelta** **AFOR** Fast-PFor	BP128[Lemire and Boystov 2013] Group-PFor Group-AFOR	● Apply split selection (AFOR,) or bit width selection (Packed Binary) on a quarter of the original integer array.

We select *Simple-9*, *Elias Gamma*, *Group Variable Byte*, *AFOR* and *PForDelta* for study. Simple-9 and AFOR algorithms have a good trade-off between fast encoding/decoding speed and high compression ratio; Group Variable Byte and PForDelta have fast encoding/decoding speed but relatively low compression ratio. These algorithms are commonly used in practice and are the representative of these four categories. The ability for our method to be instantiated in all these categories of algorithms indicates the wide applicability of the proposed framework.

4. GROUP-SIMPLE ALGORITHM

In this section, we extend the well-known Simple algorithm to a novel algorithm called *Group-Simple*, which uses the general framework in Section 4. Similar to Simple-9/16, Group-Simple still uses four bits to represent a control pattern. The difference is that a pattern in Group-Simple instructs the compression of 128-bit data rather than 28-bit data. The encoding/decoding operation of a 128-bit data segment can be potentially optimized by the 128-bit XMM register with the SIMD instructions.

4.1 Encoding Format and Storage Layout

In this part, we first introduce the storage layout and optional patterns in Group-Simple algorithm.

4.1.1 Storage Layout

In Simple-9/16 algorithms, a 32-bit codeword is divided into two parts: a 4-bit encoding pattern and 28-bit compressed data. We have a similar encoding format and aggregate two 4-bit patterns into one byte as Group Variable Byte algorithm does [Dean 2009]. Each pattern corresponds to a 128-bit data segment (i.e. four 32-bit words), which is further divided into four 32-bit sub-segments. A data segment is referred to as a *block*.

In Figure 3, we present a schematic diagram for the storage layout in Group-Simple. The major difference between Group-Simple and Simple-9/16 is that pattern segments and data segments are stored separately in physical space to achieve instruction-level vectorization: the control area stores all pattern segments, and the data area stores all data segments. In implementation, four sub-segments in a data segment share the same pattern segment. The data segments adopt the 4-way vertical storage layout.

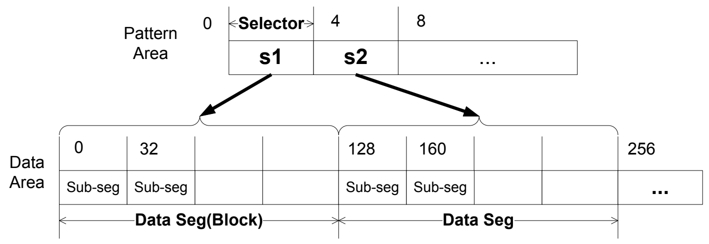


Fig. 3. Overall storage layout in Group-Simple.

4.1.2 Encoding Patterns

In Group-Simple, we use four bits to represent the encoding pattern. Each pattern corresponds to a triplet (*SEL*, *NUM*, *BW*), where *SEL* denotes the selector identifier for the optional patterns, *NUM* denotes the number of integers encoded in a 32-bit sub-segment, and *BW* denotes the bit width of an integer in the sub-segment.

Table II shows the ten optional patterns in Group-Simple. With the increase of the selector identifier (*SEL*), the number of encoded integers (*NUM*) in a sub-segment decreases, and the bit width (*BW*) of each encoded integer increases. One advantage of Group-Simple over Simple-9/16 is that the maximum bit length for an encoded integer can be up to 32 bits, which is important for compressing massive document collections with extremely large *docID*s.

Table II. Optional patterns in Group-Simple.

SEL	0	1	2	3	4	5	6	7	8	9
NUM	32	16	10	8	6	5	4	3	2	1
BW	1	2	3	4	5	6	8	10	16	32

4.2 Encoding Procedure

In this part, we first present the overall encoding procedure and then explain the key points of the encoding procedure in details. Assume that the number of integers in the original array is $4m$, where m is a positive integer. The encoding procedure of Group-Simple contains four major steps:

1) *Generating the quad max array*: Identify m quad max integers from the original $4m$ 32-bit integers (i.e. *SrcArr*) and build the array *MaxArr* with the m quad max integers.

2) *Generating of encoding patterns*: Run the pattern selection algorithm in Algorithm 1 on *MaxArr* and obtain an array *ModeArr* of the encoding patterns.

3) *Storing the encoding patterns*: Write each selector from *ModeArr* with four bits into the destination array *DstArr*. We store the start offset of data area at the head of *DstArr*. The data area is stored right after the control area.

4) *Storing the encoded data*: For each selector in *ModeArr*, we encode a group of corresponding integers by the selector pattern into a 128-bit data segment in *DstArr*.

The above encoding procedure is similar to that of the Simple algorithm except that it includes an additional step to generate the quad max array. In what follows, we will explain three key points of the encoding procedure in details.

-- Generation of the quad max array. Group-Simple adopts the 4-way vertical layout, and each four integers respectively from four consecutive sub-segments share the same encoding pattern. It indicates that all these four integers in a group share the same bit length, which is determined by the maximum integer in the group. We refer to this integer as a *quad max integer*. The *quad max array* stores all the quad max integers. Note that the quad max array is built and maintained in RAM, and will be released once we have generated the encoding patterns for all integers.

ALGORITHM 1. The pattern selection algorithm.

Input: *MaxArr, N*
Output: *ModeArr, TotalModeNum*

```
1.  ModeArr = [ ];
2.  TotalModeNum = 0;
3.  MaxArrPtr = MaxArr;    /* get the head pointer of MaxArr*/
4.  ModeArrPtr = ModeArr;  /* get the head pointer of ModeArr*/
5.  while $N > 0$ do
6.      for $i = 0$ to 9 do
7.          $(n, b) = (NUM_i, BW_i)$;  /* $NUM_i$ and $BW_i$ corresponds to the $i$th selector*/
8.          $mask = (1 << b) - 1$;     /* mask denotes the largest number with $b$ effective bits*/
9.          $pos = 0$;
10.         while $pos < n$ AND $MaxArrPtr[pos] \le mask$
11.             $pos = pos + 1$;
12.         end
13.         if $pos == n$ then break
14.     end
15.     $N = N - n$;
16.     $MaxArrPtr = MaxArrPtr + n$;
17.     $*ModeArrPtr = i$;
18.     $ModeArrPtr = ModeArrPtr + 1$;
19.     $TotalModeNum = TotalModeNum + 1$;
20. end
```

-- Pattern selection algorithm. Similar to Simple-9/16, a naïve way to select the encoding patterns is to enumerate all the integers in a sequence. Here we propose a novel pattern selection algorithm based on the quad max array, which only needs to process a quarter of the original integers. We present the pattern selection algorithm in Algorithm 1. It requires two input parameters (i_1) *MaxArr*, the head pointer of the quad max array and (i_2) *N*, the length of *MaxArr*. The algorithm returns with (o_1) *ModeArr*, the head pointer of the array of generated selectors and (o_2) *TotalModeNum*, the length of *ModeArr*. At each iteration, it examines each of the ten optional patterns in an ascending order of the selector identifiers (*SEL*) and tries to find a pattern to fit the remaining quad max integers as many as possible. With the increase of the selector identifiers, the number of integers in a data segment decreases and the bit length increases. More specially, Line 1~4 are the initialization steps for variables, Line 6~13 are the inner loop for pattern selection, and Line 15~19 are the update steps for the variables. It is worth explaining the inner loop for pattern selector in more details. The loop in Line 6 breaks when (1) the number of integers reaches the limit of the selector, i.e., *NUM* and (2) a quad max integer cannot be held in *BW* bits. We use the shift operation to obtain the largest number with an effective bit length of *b* bits, i.e., the variable *mask*.

– **Packing original integers into blocks.** After pattern selection, we examine how to pack a sequence of integers into blocks with the generated encoding patterns. To encode a single block, the algorithm loops *NUM* times. At each time it uses the sequential shift and mask operations to store four integers respectively into four sub-segments, where *NUM* is the number of compressed integers in a sub-segment. And the algorithm will encode 4**NUM* integers into a 128-bit data segment.

For better understanding the above encoding procedure, Figure 4 presents an illustrative example for the encoding of a sequence of twenty 32-bit integers by using Group-Simple. The quad max integers are marked in bold. To hold the largest integer 64, we need at least a bit width of 6 bits. Therefore, the 5th selector in Table II is selected, which indicates that 5 integers are packed in a 32-bit sub-segment and each encoded integer occupies 6 bits.

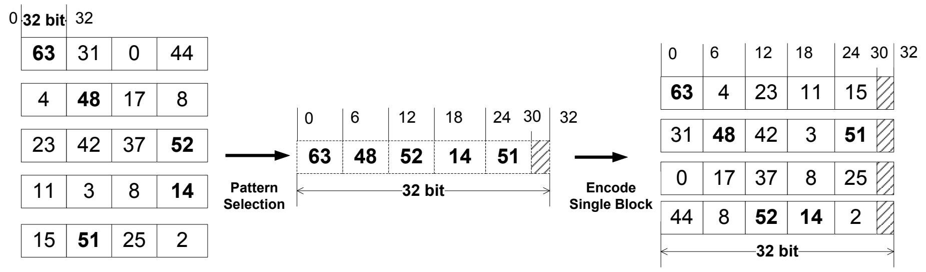


Fig. 4. An example to illustrate the encoding procedure of Group-Simple.

4.3 The Decoding Procedure

The key step in decompression is to decode a single block, which is a 128-bit data segment of 4**NUM* encoded integers (*NUM* is the number of integers in a sub-segment corresponding to the selector of this block). In this procedure, the algorithm loops *NUM* times, and in each loop we use the sequential shift and mask operations to decode four integers respectively from four sub-segments.

We present the decoding procedure of Group-Simple as follows:

1) Read the start offset of the data area from the head pointer of *SrcArr*, and locate the start position of the control area, denoted by *ModePos*. The start position of the data area is denoted as *DataPos*.

2) Read four bits from *ModePos* and obtain the current selector.

3) Decode a 128-bit data segment starting at *DataPos* with the current selector.

4) Move *ModePos* and *DataPos* forward by 4 bits and 128 bits respectively. If *ModePos* does not reach the end, go to step 2.

4.4 SIMD-based Implementation and Optimization Tricks

The SIMD-based implementation of Group-Simple is called as *SIMD-Group-Simple*. Once we have transformed the sequential layout into the format in Figure 3, it is relatively easy to apply SIMD instructions to implement SIMD-Group-Simple as described in Section 4.6.

Here we mainly discuss the optimization tricks in practice. Conditional statements would affect the use of instruction pipelines [Schlegel et al. 2010]. In the encoding procedure, the generation of the quad max array involves conditional statements for value comparison (i.e. identify the maximum value from four integers). The function of the quad max array is to determine the suitable bit widths for encoded integers. To reduce conditional statements, we do not need to identify the exact quad max integers but the *pseudo quad max integers* instead. Following [Lemire and Boystov 2013], we use the logical *OR* operations to generate pseudo quad max integers, which may not be equal to the real quad max integer but have the same *effective bit width*. When decoding a single block, since both *SEL* and *NUM* have a fixed set of optional values, we replace the conditional statements with *SWITCH-CASE* statements in the loop. The above tricks yield 20% and 50% improvement at the encoding and decoding speed respectively by reducing conditional statements.

5. THE GROUP-SCHEME COMPRESSION ALGORITHM FAMILY

In the previous section, we have presented the instantiation of Simple algorithm in the proposed framework. Inspired by two well-known algorithms Elias Gamma [Elias 1975] and Group Variable Byte (GVB) [Dean 2009], we will present another family of new compression algorithms in the framework, called *Group-Scheme*. Group-Scheme has borrowed the key idea both from GVB and *k*-Gamma [Schlegel et al. 2010], and generalizes it by incorporating compression granularity and length descriptor. Specially, as will be shown later, *k*-Gamma can be considered as a variant in our Group-Scheme family.

5.1 Variants in Group-Scheme family

To better describe the variants in Group-Scheme, we first introduce two terminologies: compression granularity and length descriptor.

- **Compression granularity** (*CG*) is defined as the minimum unit operated on (or allocated) by a compression algorithm. For example, the compression granularity of Elias Gamma and *k*-Gamma coding is 1 bit, while the compression granularity of Variable Byte encoding is a byte, i.e. 8 bits.

- **Length descriptor** (*LD*) is defined as the minimal number of compression granularities to represent (or encode) an integer. Length descriptor itself can be either binary coded or unary coded. For instance, given the compression granularity of 2 bits, the length descriptor of an integer 458_{10} (111001010_2) has the value 5 (we need five 2-bit compression units to hold ten bits), which can be represented as "101" in binary or "11110" in unary.

We set up four compression granularities for Group-Scheme: 1, 2, 4 and 8 bit(s). By combining optional values of compression granularities and length descriptors, the Group-Scheme family contains eight different variants in total as summarized in Table III.

Table III. Algorithm variants in Group-Scheme famliy.

Length Descriptor (LD)	**Compression Granularity (CG)**			
	1 bit	**2 bits**	**4 bits**	**8 bits**
Binary	5	4	3	2
Unary	1~32	1~16	1~8	1~4

In fact, k-Gamma [Schlegel et al. 2010] encoding can be considered as a special case of our Group-Scheme family, i.e., when the compression granularity is set to one bit and the length descriptor adopts the complete unary coding, Group-Scheme becomes k-Gamma.

5.2 Encoding formats and encoding/decoding procedure

5.2.1 Encoding format

Group-Scheme follows the format of data area described in Section 4, and the major difference lies in the control area, which is composed of several encoded length descriptors. A length descriptor corresponds to a pattern segment in the framework. Based on the coding type (binary/unary), Group-Scheme stores length descriptors (*LD*) in two ways:

(a) *Binary LD:* Figure 5 shows four different encoding formats for control area with binary length descriptors. The maximum number of bits for LD is determined as follows

$$\lceil \log_2(32/\mathrm{CG}) \rceil = 5 - \lceil \log_2 CG \rceil .$$

In order to avoid the cross-byte or cross-double-byte decoding, we adopt the aligned storage at the cost of several empty bits.

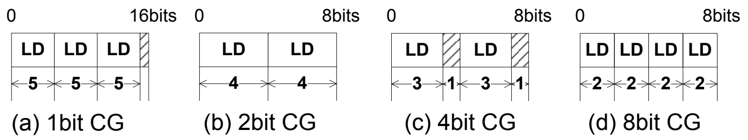


Fig. 5. The corresponding formats of Group-Scheme with different binary length descriptors. The compression granularity and length descriptor are shortened as CG and LD respectively.

(b) *Unary LD:* There are two methods to store unary LDs. If a unary length descriptor can be stored across bytes, we call it *Complete Unary*[1] or *CU* for short. As a comparison, if length descriptors cannot be stored across bytes, we call it *Incomplete Unary* or *IU*. Note that we do not consider the 1-bit and 2-bit CG for unary LD, since the maximum number of bits needed for a single LD will exceed eight bits with 1-bit and 2-bit CG. Figure 6 shows the example of IU and CU for the 4-bit compression granularity.

[1] Following [Stepanov et al. 2011], we use the terminologies of complete unary and incomplete unary to discriminate between cross-byte storage and byte-aligned storage.

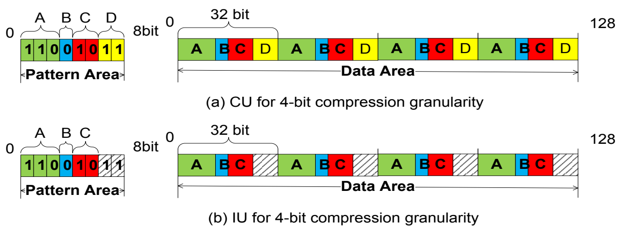


Fig. 6. An example to compare incomplete unary coding (IU) and complete unary coding (CU).

5.2.2 Encoding procedure

At each time, we first find the maximum integer $quadmax$ of four consecutive integers, and then calculate the length descriptor for $quadmax$ as follows

$$\text{ValueOfLD} = \begin{cases} \lceil \lceil log_2(quadmax+1) \rceil / CG \rceil - 1, & Binary\ LD \\ \lceil \lceil log_2(quadmax+1) \rceil / CG \rceil, & Unary\ LD \end{cases} \quad (1)$$

The length descriptor (either unary or binary) is stored in control area. We use the sequential shift and mask operations to encode four integers by taking the low $\lceil \log_2(quadmax+1)/CG \rceil \times CG$ bits of an integer. Furthermore, these four encoded integers are stored into four different sub-segments respectively. We update the pointer for the sub-segment and the bit offset within the current sub-segment. The above steps are repeated until all the integers are encoded.

5.2.3 Decoding procedure

At each time, we first read a length descriptor from the control area and calculate the bit width BW for the encoded integers of data area, where

$$BW = \begin{cases} CG \times (ValueOfLD + 1), & Binary\ LD \\ CG \times ValueOfLD, & Unary\ LD \end{cases}. \quad (2(2)$$

We loop four times to extract four BW-bit integers (probably with leading zeros) respectively from four consecutive 32-bit sub-segments. Then we update the pointer for the sub-segment and the bit offset within the current sub-segment. The above steps are repeated until all the integers are decoded.

5.3 SIMD-based Implementation and Optimization

Following Section 4, we can easily implement the SIMD-based version of Group-Scheme in the proposed framework, i.e. SIMD-Group-Scheme, which vectorizes the encoding/decoding operations for four consecutive integers in a data segment. In this part, we will further present several optimization tricks for efficient implementation of Group-Scheme and SIMD-Group-Scheme.

5.3.1 Packed decoding technique for length descriptors

Our experiments empirically showed that the main bottleneck of the decoding procedure lies in recovering the length descriptors. It becomes even worse for unary-coded length descriptors since we have to examine whether the current bit is the end of a length descriptor by using the condition statements. To alleviate this problem, we propose to use the packed decoding technique [Lemire and Boystov 2013] to decode control patterns as described below:

(a) *Unary LD:* At each time, we read a byte instead of a bit and decode all length descriptors in the byte. We compile a lookup table to speed up the decoding process. An 8-bit unary sequence totally has $2^8 = 256$ possibilities. Corresponding to the 256 possibilities, we generate all necessary decoding information and store the information with an array of 256 *STRUCT* elements. Each element contains the following information: (1) the number of encoded integers, (2) the bit width of an encoded integer, and (3) the length of the last consecutive "1"s subsequence without an ending zero (i.e. in the format of "11…1"). For example, a unary sequence "10110011" (left to right) can be decomposed into four parts: "10", "110", "0" and "11". We can obtain three length descriptors: 2, 3 and 1. Note that the last subsequence "11" contains two 1s but does not end with a 0. To deal with such cross-byte length descriptors, we record the incomplete part, and insert it at the beginning of the next 8-bit sequence. We also experimented with the 16-bit unary sequence which has 65,536 possibilities, but not notice significant difference on performance.

(b) *Binary LD*: Similar to unary LD, we can use the similar packed decoding technique by using the lookup tables. An extra type of information is needed: the total number of bits actually stored in a sub-segment. This information can help determine the current pointer for the sub-segment and bit offset within the sub-segment.

Based on our experiments, the *packed decoding technique* for length descriptors can yield about 50% improvement at the decoding speed for most algorithms in Group-Scheme family.

5.3.2 SIMD-based Group Unpacking in Data Area for SIMD-Group-Scheme

We further discuss the optimization tricks for SIMD-Group-Scheme in implementation. In the above, we have used the packed decoding technique to simultaneously decode several length descriptors. Now we apply it to data area and decode $4m$ integers at each time, where $m > 1$.

We improve the packed decoding technique by using SIMD instructions. For each 8-bit pattern, we have generated a sequence of SIMD assemble instructions to decode the corresponding integers correspondingly. The above optimization trick is effective to reduce the updating operations of bit offset for vectorized shifting right/left instructions. Next we describe the implementation details of these assemble functions respectively for unary and binary LD.

(1) *Unary LD:* Assume that there are $4m$ integers to be decoded according to an 8-bit unary pattern in control area, where $m \geq 1$. If the length descriptor is complete-unary coded, a XMM register will be used to keep the number of unprocessed 1s in the last 8-bit unary sequence in control area (called XMM1), and another XMM register (called XMM2) is used to keep the corresponding four incomplete integers of the last data segment in data area. The bit offset in the sub-segment is stored in XMM3. Now the bit offset remains in the register and only need to be updated after decoding the $4m$ integers. The steps in the assemble function are as follows:

[Step 1] Load several 128-bit segments to be decoded into XMM registers.
[Step 2] Make all sub-segments word-aligned by using logical right/left-shift SIMD instructions with the bit offset in XMM3.
[Step 3] Decode the first four integers in the sub-segments. Left-shift these four integers by the value in XMM1 and execute vectorized bitwise OR with XMM2. Then the first four integers are decoded and written to memory.

[Step 4] Use the vectorized shift and mask operations to decode the rest 4*(m-1) integers and write them back to memory.
[Step 5] Write the number of unprocessed unary bits into XMM1, and the corresponding four incomplete integers from data area into XMM2.
[Step 6] Update the bit offset in XMM3.

Note that the above steps are specially designed for complete-unary length descriptors. When using incomplete unary coding, the case becomes simpler: only Step 1 and Step 4 are needed.

(2) *Binary LD:* Similarly, assume that there are 4m integers to be decoded according to an 8-bit pattern sequence. When the length descriptor is binary coded, the function structure is also simpler: Step 3 and 5 can be removed.

In our implementation, the above optimization tricks can yield about 30-100% improvement in the decoding speed for SIMD-Group-Scheme.

6. INSTATIATION OF THE FRAMEWORK ON FRAME FIXED-BIT LENGTH ALGORITHMS

In this section, we mainly discuss about the application of our SIMD-based compression framework on the fourth category of compression algorithms, which pack the integers with a fixed bit width in a split, including AFOR and PForDelta. A split of integers with the same bit length is referred to as a *frame*. We call the instantiated algorithms as Group-AFOR and Group-PFD respectively.

6.1 Group-AFOR

The group-based algorithm Group-AFOR is a modification of Adaptive Frame of Reference (AFOR) in [Delbru et al. 2012]. Group-AFOR partitions a sequence of integers into multiple frames of variable lengths. The frame length, which is also known as the frame size, is further restricted to three optional values {32,64,128}. The optimal configuration of frame partition and frame lengths is solved by an efficient dynamic programing algorithm. The major difference is that we incorporate the quad max array to speed up the partition step: only a quarter of integers are needed to process. Note that although the algorithm runs on the quad max array, the cost is computed based on the original integer array. After the partition step, we use the 4-way vertical layout to encode each frame of the original array.

A similar algorithm to consider here is VSEncoding algorithm [Silvestri and Venturini 2010]. The major difference between AFOR and VSEncoding lies in the number of optional frame lengths: AFOR provides three lengths while VSEncoding provides five lengths. We implemented the group based VSEncoding in our framework, and the findings were similar to those of Group-AFOR. Therefore, we omitted VSEncoding in this paper.

6.2 Group-PFD

The major difficulty in wrapping PForDelta is how to identify the exceptional integers in the original PForDelta algorithm with the quad max array. We first examine the exceptional entries on the quad max array. Once an exception in the quad max array has been found, we will further examine the other three integers in the corresponding group from the original array.

The detailed procedure for the SIMD-based version of PForDelta (i.e. SIMD-Group-PFD) is described as follows:

[Step 1] Generate the max quad array *MaxArr*.

[Step 2] Run the procedure of calculating the bit width in PForDelta on *MaxArr* : for each block in *MaxArr*, identify the smallest bit width b such that the exception ratio of the block is below a given threshold ζ ($0< \zeta <1$). Meanwhile, record the positions of the exceptions in an array *ExOffsetArr*.

[Step 3] According to the bit width of each block and *ExOffsetArr*, generate the normal array and exception array based on the original integer sequence: for each exception position in *ExOffsetArr*, we further examine whether each of the other three integers in the same group is an exception. We update the exception array with the new exceptional positions.

[Step 4] First, apply SIMD instructions to encode the normal array with the vertical storage layout in the data segment. Then, encode the exception array by following the approach in [Zhang et al. 2008], which will be encoded with three effective bit widths 8 bits, 16 bits and 32 bits respectively.

The decoding procedure is described as follows, which is similar to original PForDelta:

[Step 1] Read the bit width and first exception offset for each block.

[Step 2] Use SIMD instructions to decode the normal array.

[Step 3] Use SIMD instructions to decode the exception array and write each exception into the corresponding position of the normal array.

6.3 Connections with SIMD-BP128

Lemire et al.'s SIMD-BP128 [Lemire and Boytsov 2013] aggregated 128 consecutive integers as a frame and adopted the 4-way vertical layout. SIMD-BP128 packs 128 integers with a fixed bit width, which is the minimum number of bits to hold the largest integer among these 128 integers. SIMD-BP128 can be naturally fit into our compression framework with slight modifications. First, find 32 quad max integers for each 128 integers, and then identify the largest quad max from these 32 quad max integers and compute the corresponding effective bit width. The above steps can be regarded as a special case of Step 1 and 2 in Section 4. For SIMD-BP128, the frame size is fixed as 128, which may not be flexible to tune on different datasets. In specific, it needs to pay for more extra space for a large frame size: SIMD-BP128 has a low compression ratio (See Section 8). While our framework is more flexible to support various frame sizes, SIMD-BP128 can be considered as a special variant in the proposed framework.

7. EXPERIMENTS

In this section, we first introduce the experimental settings, and then evaluate the proposed compression algorithms with several state-of-the-art algorithms. The evaluation metrics concern the following three aspects respectively: encoding speed, decoding speed and compression ratio. Finally, we examine the performance of different algorithms by the time cost (i.e. query processing performance) and space cost (i.e. index size) in a laboratory search engine.

7.1 Experimental Settings

(1) **Server profile.** Our experiments are tested on a server with an Intel Core i5-3470S processor (2.9GHz, 12MB L3 Cache) and a 8GB 1333MHz RAM. The operating system is the 64-bit Linux of version 2.6.32-71. All the compression algorithms are implemented in C++ and complied by using "g++ 4.6" with the optimization flag "-O3".

(2) **Test collections**. We test the algorithms on several datasets representing different characteristics. The basic statistics of the four datasets are shown in Table IV. The first two datasets, TREC GOV2 and TREC Clueweb09B, are released by TREC, which are the standard test collections for the evaluation of the index and retrieval systems. The Wikipedia and Twitter datasets are mainly used to test the algorithm stability on various datasets. Wikipedia[2] is well known for providing formal definitions and knowledge on concepts and entities; Twitter is the most representative microblogging service, and we use the shared dataset in [Kwak et al. 2010]. For Twitter collection, we aggregate all the tweets of a user as a document (a.k.a. *user document*), and treat the text in a user profile as the title of the "user document". For these four datasets, we have extracted the title and body fields of each document, and then built an inverted index for each dataset with a search engine system implemented by our group [Shan et al. 2012]. We find that over 90% of d-gap and TF on all four datasets can be represented in 8 bits, which indicates the potential compressibility of these four datasets.

Table IV. Basic statistics of the four datasets (1M=1,000,000) .

Statistics	**GOV2**	**ClueWeb09B**	**WikiPedia**	**Twitter**
# Documents	25M	50M	10M	9M
# Terms	55M	88M	46M	31M
# Postings	5,249M	11,975M	1,338M	1,466M
# Tokens	19,446M	28,796M	3,441M	3,575M

In what follows, we first present the evaluation experiments on the standard TREC datasets, and then further validate the performance of different algorithms on Wikipedia and Twitter datasets. For TREC datasets, we first randomly selected 10,000 queries from the query set of 2005 TREC Terabyte Track[3], and then kept all the unique terms from these selected queries by excluding stopwords. Next we extracted the d-gap sequences and TF sequences from the posting lists of these selected terms, and performed the encoding/decoding operations using different compression algorithms. Finally, we average the performance over all the selected terms on encoding speed, decoding speed and compression ratio as the final results.

(3) **Evaluation metrics.** The results will be presented in the order of decoding speed, encoding speed and compression ratio. In order to reduce the system disturbance, the final result is the average of ten runs. We use *the number of million integers per second* (*mis*) as the speed metrics, and a larger value indicates better performance. We use *the average bits needed per encoded integer (bits/int)* as the metrics of compression ratio (rate), and a smaller value indicates better performance.

(4) **Methods to compare.** We compared the proposed algorithms with several state-of-the-art compression algorithms. An algorithm named with the prefix "SIMD-" indicated that it has been optimized by SIMD instructions in the encoding/decoding procedure. For both the sequential and SIMD-based algorithms, we have adopted the similar optimization tricks as described in previous sections. We present the comparison algorithms as follows:

– *PForDelta*. PForDelta has been one of the most competitive sequential compression algorithms in terms of encoding/decoding speed. Our implementa-

[2] http://dumps.wikimedia.org/enwiki/
[3] http://trec.nist.gov/data/terabyte/05/05.efficiency_topics.gz

tion of PForDelta adopted the optimization method for compressing the exception values in [Zhang et al. 2008]. The exception ratio is set to 10% (SIMD-Group-PFD also follows the same setting).

– PackedBinary and SIMD-FastPFor [Lemire and Boystov 2013]. We set the block size to 512 bits and use the open-source implementation on GitHub.[4]
– Our proposed algorithms include: Group-Simple, Group-Scheme, Group-AFOR, Group-PFD and their corresponding SIMD based versions. As previously mentioned, SIMD-BP128 can be considered as a special variant of the proposed framework, thus we have re-implemented SIMD-BP128 with slight modifications in our framework and take SIMD-BP128 as one of our algorithms, too.

Since Group-Scheme contains many variants, we first examine various combinations of compression granularity and length descriptor, and only keep the variants with better performance in following experiments.

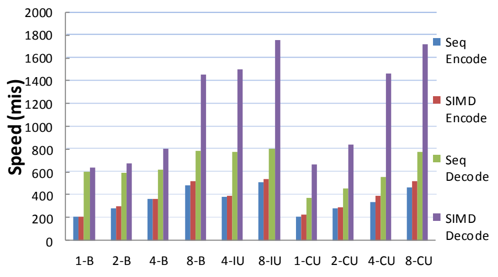
7.2 Variant Selection in the Group-Scheme Family

Group-Scheme variants are named in the pattern of “CG-LD”, where the compression granularity *CG* can be set to 1/2/4/8 bit(s) and the length descriptor *LD* can be set to B/IU/CU. In particular, *k*-Gamma (*k*=4) can be considered as a special variant of Group-Scheme, i.e. the variant “1-CU”. The results of encoding/decoding speed and compression ratio are shown in Figure 7(a) and Figure 7(b) respectively.

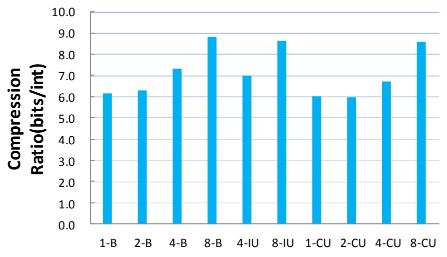
We first analyze the results of decoding speed in Figure 7(a). First, we observe that the SIMD-based algorithms significantly outperform the corresponding sequential algorithms with a large margin varying from 40% to 110%, which indicates the effectiveness of SIMD-based vectorization. Second, for both the sequential and SIMD-based variants, large compression granularities lead to good performance. The main reason is that large compression granularities correspond to smaller bit widths for LDs, thus the packed decoding technique can process more LDs in an 8-bit pattern sequence. Third, the SIMD-based variants with unary-coded LD (e.g. SIMD-based 4-CU) are faster than those with binary-coded LD (e.g. SIMD-based 4-B) by 20% to 50%. The variant SIMD-based 8-IU achieves the fastest decoding speed, and has a relatively low compression ratio. For the encoding speed, we have similar observations with decoding speed, but the improvement by incorporating SIMD-based vectorization is small.

We continue to analyze the results of compression ratio in Figure 7(b). With the increase of compression granularity, we can see the compression ratio becomes worse. Smaller CU stores the encoded integers more compactly and reduces more empty bits in data area, but the corresponding length descriptors take up more space in control area. There is a trade-off between these two types of space cost. In our experiments, the former is the dominant factor for space cost. In addition, the unary-coded LD consistently leads to better compression ratio than binary-coded LD on all compression granularities.

[4] http://github.com/lemire/FastPFor



(a) Decoding/Encoding speed



(b) Compression ratio

Fig. 7. Performance comparison on compressing d-gaps of GOV2 within Group-Scheme.

For Group-Scheme, variants with larger compression granularity (4 or 8 bits) have faster encoding/decodng speed, and variants with smaller compression granularity (1 or 2 bits) have better compression ratio with competitive encoding/decoding speed. Based on the above findings, we select four competitive variants and their corresponding SIMD-based implementation, including Group-Scheme 1-CU (simplified as *GSC-1-CU*), *GSC-8-IU* and SIMD-Group-Scheme 1-CU (simplified as *SIMD-GSC-1-CU*) and *SIMD-GSC-8-IU*.

7.3 Evaluation on the Compression of Posting Lists

In this part, we evaluate the performance of compressing d-gaps and TF sequences of posting lists. To save space, our Group-AFOR, Group-Simple and Group-PFD algorithms will be simplified as G-AFOR, G-SIM and G-PFD respectively in the following tables. We first present the results on GOV2 and ClueWeb09B datasets in Table V and Table VI, and then present the results on Wikipedia and Twitter datasets in Table VII. For all tables, we organize the results of algorithms by four categories. Note that we consider Group-Scheme as *bit-aligned* since it originates from the idea of Elias Gamma.

Table V. Comparison of decoding/encoding speed on GOV2 and ClueWeb09B (mis).

Category	Algorithm	Decoding Speed				Encoding Speed			
		GOV2		ClueWeb09B		GOV2		ClueWeb09B	
		d-gap	TF	d-gap	TF	d-gap	TF	d-gap	TF
Bit-aligned	Rice	67	81	71	84	60	74	61	86
	Gamma	50	84	58	101	63	85	66	97
	GSC-8-IU	801	809	740	801	509	516	458	516
	GSC-1-CU	376	443	364	475	211	232	205	248
	SIMD-GSC-8-IU	1756	1772	1546	1635	534	552	497	552
	SIMD-GSC-1-CU	463	777	534	969	222	243	217	263
Byte-aligned	VarByte	538	655	519	671	518	627	495	641
	GVB	514	503	510	506	260	265	253	270
	G8IU	531	537	508	545	148	121	126	130
	G8CU	489	510	464	507	146	119	124	124
	SIMD-GVB	823	831	823	823	252	270	247	267
	SIMD-G8IU	1695	1719	1501	1613	149	125	127	130
	SIMD-G8CU	1282	1325	1243	1285	145	123	128	127
Word-aligned	Simple-9	522	644	506	743	93	114	90	145
	Simple-16	484	613	500	744	49	61	52	82
	G-SIM	943	1129	1009	1266	183	208	178	232
	SIMD-G-SIM	1855	1954	1577	1922	242	237	204	265
Frame	PackedBinary	1392	1392	1252	1359	244	244	237	244

fixed bit length	PForDelta	1186	1048	1028	999	36	42	33	50
	AFOR	725	579	672	569	247	208	195	211
	G-AFOR	804	756	643	779	242	224	212	223
	G-PFD	1194	1057	1020	1007	89	102	80	120
	SIMD-BP128	2273	2049	1671	2155	592	539	318	792
	SIMD-FastPFOR	1912	1692	1258	1405	188	130	139	134
	SIMD-G-AFOR	1976	1819	1440	1673	368	354	366	367
	SIMD-G-PFD	2126	1711	1695	1543	220	210	170	192

7.3.1 **Decoding Speed**

In Table V, it is clear to see that nearly all SIMD-based group algorithms (e.g. SIMD-Group-Simple) outperform the corresponding non-SIMD group algorithms (e.g. Group-Simple) and original sequential algorithms (e.g. Simple). The major observations are listed as follows:

(1) In the bit/byte-aligned category, Rice and Elias Gamma have very slow decoding speed. Overall, GVB has relatively better decoding speed, but it is still much lower than our proposed GSC-8-IU.

(2) In the word-aligned category, Group-Simple is much faster than traditional Simple algorithms (1.5~2.5 times). The major reason is that Group-Simple relates a control pattern to four 32-bit sub-segments while Simple-9/16 relates a control pattern to only one 28-bit data segment. The incorporation of the shared pattern will also improve the hit rate of CPU cache. Furthermore, SIMD-Group-Simple is significantly faster than Group-Simple (1.5~2 times) and traditional Simple algorithms (3~3.8 times).

(3) In the frame fixed bit-width category, SIMD-BP128 has achieved the fastest decoding speed, which is similar to the finding in [Lemire and Boytsov 2013]. As discussed in Section 7.2, SIMD-BP128 can be considered as a special variant of our compression framework. Following SIMD-BP128, SIMD-Group-PFD and SIMD-FastPFor also achieve very competitive decoding speed. SIMD-Group-AFOR is two times fast as Group-AFOR, and Group-AFOR is slightly faster than AFOR.

7.3.2 **Encoding Speed**

Compared to decoding speed, encoding speed is less important than decoding speed since the index is usually built offline. Here we mainly want to examine whether the incorporation of SIMD instructions can accelerate the encoding procedure. The results are shown in the last four columns of Table V.

Overall, we observe that SIMD based implementation leads to some improvement in encoding speed, e.g., SIMD-Group-AFOR is faster than Group-AFOR and AFOR by nearly 50%, but the improvement is relatively smaller than that for decoding speed. In addition, our group based algorithms have faster encoding speed: (a) Group-Scheme variants are faster than GVB and Elias Gamma by 3~4 times; (b) Group-Simple is faster than Simple-9 (1.5~2 times) and Simple-16 (2.5~4 times). The key reason for improvement lies in the use of the quad max array. We only need to process a quarter of all the integers with the help of the quad max array.

7.3.3 **Compression Ratio**

Table VI shows the comparison of compression ratio. Since the incorporation of SIMD instructions does not affect compression ratio, we only present the results of non-

SIMD algorithms. Overall, group based algorithms have relatively lower compression ratio compared to the corresponding sequential algorithms. The main reason is that bit length is determined by the maximum integer in a group, which will generate more "empty" bits. In order to achieve better vectorization, our algorithms need to pay extra space but still have competitive compression ratios, e.g., GSC-1-CU is slightly worse than the best baselines Rice and Gamma (in terms of compression ratio) but they are 6~10 times faster than Rice and Gamma.

Table VI. Comparison of compression ratio on GOV2 and ClueWeb09B (bits per integer).

Category	**Algorithm**	**GOV2**		**ClueWeb09B**	
		d-gap	**TF**	**d-gap**	**TF**
Bit-aligned	**Rice**	**5.0**	**3.1**	**5.1**	**2.4**
	Gamma	6.7	2.8	4.8	2.2
	GSC-8-IU	8.6	4.9	8.8	8.3
	GSC-1-CU	6.0	3.3	4.9	2.5
Byte-aligned	VarByte	8.3	8.0	8.3	8.0
	GVB	10.1	10.0	10.2	10.0
	G8IU	9.2	9.0	9.2	9.0
	G8CU	9.2	9.0	9.2	9.0
Word-aligned	Simple-9	6.3	4.0	5.3	3.1
	Simple-16	5.9	3.7	5.0	2.9
	G-SIM	6.5	4.8	6.2	3.8
Frame fixed bit-length	PackedBinary/BP128	7.0	7.1	8.8	6.1
	PFORDelta	5.9	4.7	6.2	4.3
	FastPFor	5.8	3.6	5.4	2.9
	AFOR	6.0	4.0	5.3	3.1
	G-AFOR	6.1	4.7	6.0	3.6
	G-PFD	6.2	5.1	6.8	4.5

7.3.4 **Results on Wikipedia and Twitter datasets**

We continue to evaluate different algorithms on Wikipedia and Twitter datasets, which are used to examine the algorithm stability. Due to space limit, we only report the results on d-gaps, and the results on TFs have similar findings.

In Table VII, we can observe that our proposed algorithms still work well on these two datasets: 1) Overall, our SIMD-based algorithms have faster encoding/decoding speed and slightly worse compression ratio than the corresponding sequential-version algorithms. 2) Rice have best compression ratio. 3) SIMD-BP128 has the best decoding speed followed by another two competitive algorithms SIMD-Group-PFD and SIMD-FastPFor. 4) Frame fixed bit-width algorithms have similar compression ratio except that SIMD-BP128 has a much lower compression ratio.

Table VII. Performance comparsion on d-gaps of Wikipedia and Twitter Datasets.

Category	**Algorithm**	**Decoding Speed** (mis)		**Encoding Speed** (mis)		**Compression Ratio** (bits per integer)	
		Wiki	**Twitter**	**Wiki**	**Twitter**	**Wiki**	**Twitter**
Bit-aligned	Rice	67	67	60	60	**5.4**	**5.5**
	Gamma	48	46	61	61	7.2	7.5
	SIMD-GSC-8-IU	1936	2028	515	528	9.0	8.8
	SIMD-GSC-1-CU	445	429	218	219	6.3	6.6
Byte-aligned	SIMD-G8IU	1775	1841	144	153	9.4	9.2
Word-aligned	SIMD-G-SIM	1809	1975	230	238	7.0	7.1
Frame fixed	PFORDelta	1202	1264	35	34	6.5	6.4

bit-length	SIMD-BP128	**2003**	**2133**	**449**	**507**	7.8	7.7
	SIMD-FastPFor	2041	2210	177	195	6.2	6.3
	SIMD-G-AFOR	1810	1929	430	439	6.6	6.7
	SIMD-G-PFD	2230	2434	198	201	6.7	6.7

7.4 Evaluation on query processing performance

In the previous experiments, we have studied the decoding speed of different algorithms. A more direct comparison way is to examine the overall performance of query evaluation with different compression algorithms. The time cost for per query processing typically includes the following steps: loading posting lists from disks to memory, decoding d-gaps, decoding TFs, recovering DocID based on d-gap, locating documents with skip pointers, scoring the candidate documents and top-K documents retrieval by using heap sort. Among these steps, only the first three steps are related to compression algorithms, i.e., *loading posting lists from disks to memory*, *decoding d-gaps*, and *decoding TFs*.

In our experiments, we have found that the cost from the disk IO is large and not stable. Therefore, we follow the method of using warm cache in [Delbru et al. 2012], i.e., the time measurements are made when the part of the index read during query processing is fully loaded in memory. The query processing speed is measured by the query rate, i.e., the number of queries a system can process per second. Furthermore, we execute each query by ten times to reduce the disturbance from OS and machine, and take the average of ten runs as the final performance for a query.

We still use the GOV2 datasets and the same TREC query set described in Section 7.1 for evaluation. The query evaluation adopts the DAAT (document-at-a-time) scoring way and the top-k retrieval with k set as 10. We use the Okapi BM25 probabilistic model [Robertson et al. 1999] to measure the relevance between a candidate document and a query. Two types of queries are considered: AND query and OR query. We make use of the skipping lists for AND queries. For OR queries, we considered WAND [Broder et al. 2003; Ding and Suel 2011] and MaxScore [Jonassen et al. 2011; Shan et al. 2012].

In order to better see the advantage of SIMD-based algorithms, we present the results for **AND** query processing performance descendingly in Table VIII. We can observe 1) most of the top ten ranks are occupied by SIMD-based algorithms; 2) the proposed group compression algorithms with SIMD instructions have outperformed the sequential group algorithms. 3) The SIMD-BP128 and SIMD-Group-Simple achieve very competitive performance. In our experiments, decoding of d-gap and TF roughly takes 15%~35% of the overall time cost, therefore the improvement is less significant than that for decoding speeds in Table V.

The results on **OR** queries are similar to what have been observed for **AND** queries, but the overall performance difference between algorithms is relatively small. To save the space, we omit the results on **OR** queries.

Table VIII. Performance rankings on average query processing rate (the number of queries processed per second).

GOV2		*ClueWeb*	
Algorithms	**Query Rate**	**Algorithms**	**Query Rate**
SIMD-BP128	168.4	SIMD-BP128	68.2
SIMD-G-PFD	166.7	SIMD-G-SIM	67.9
SIMD-G-SIM	166.4	SIMD-G8IU	66.8
SIMD-FastPFor	164.2	SIMD-G-PFD	66.6
SIMD-G8IU	162.9	SIMD-FastPFor	66.2

SIMD-G-AFOR	161.8	SIMD-G-AFOR	65.1
SIMD-GSC-8-IU	160.5	PackedBinary	63.4
PackedBinary	156.0	SIMD-GSC-8-IU	62.6
SIMD-G8CU	148.1	SIMD-G8CU	60.5
G-SIM	146.0	Group-Simple	59.7
PFORDelta	143.1	PFORDelta	57.1
G-AFOR	134.4	VarByte	52.7
VarByte	130.7	Group-AFOR	52.4
AFOR	127.1	SIMD-GVB	50.7
SIMD-GVB	126.1	SIMD-GSC-1-CU	49.8
Simple-9	122.2	Simple-9	49.5
SIMD-GSC-1-CU	119.3	AFOR	49.0
GVB	118.8	Simple-16	48.9
Simple-16	118.2	GVB	47.6
G8IU	117.4	G8IU	46.9
G8CU	107.8	G8CU	42.5
Rice	32.1	Rice	13.5

7.5 **Evaluation on index sizes**

In this part, we compare the index sizes of different compression algorithms. Generally, the procedure of index building includes two major steps:

(1) Segment building: We gradually add documents to the in-memory inverted index. Once it has reached the limit of segment memory, we flush in-memory segment data to the disk. In this step, several local indices will be generated and the posting lists are not compressed.

(2) Index merging and optimizing. We merge all local indices into a global index and perform index optimization. In particular, we focus on the size of posting lists stored on the disk, which consist of three parts in our experiment: d-gaps, TFs and skip list pointers for posting blocks. The skip distance of skip list pointers is set to 512 postings. Skip lists and dictionary take very little space. Therefore the overall index size is determined by the size of encoded posting lists. An algorithm with a better compression ratio will generate a smaller index size. Our proposed SIMD-based algorithms will take up at least 128 bits no matter how small the size of posting lists is. Therefore, we adopt the traditional compression algorithms for short posting lists: posting lists with a size less than 64 are compressed by Variable Byte encoding [Scholer et al. 2002].

Table VIII. Comparison of index sizes on four datasets with different compression algorithms (MB).

Category	**Algorithm**	**GOV2**	**Clueweb09B**	**Wiki**	**Twitter**
Uncompressed	Uncompressed	40466	92062	11994	12518
Bit-aligned	Gamma	9571	16285	4479	4140
	Rice	**7280**	**15120**	**3842**	**3387**
	GSC-1-CU	8604	16144	4021	3755
	GSC-8-IU	12883	28614	5176	4988
Byte-aligned	VarByte	12096	26378	4917	4725
	GVB	14167	31382	5418	5282
	G8CU	13691	29812	5715	5356
	G8IU	13852	30149	5766	5416
Word-aligned	Simple-9	9078	17189	4236	3943
	Simple-16	8650	16315	4139	3826
	G-SIM	9629	19948	4316	4068
Frame fixed bit length	PackedBinary	10180	23355	4460	4202
	PForDelta	8699	19434	4095	3806
	AFOR	8071	15982	3870	3576

	G-AFOR	8963	18668	4118	3856

Table VIII shows the index sizes of different datasets by using different compression algorithms. Among all the algorithms, Rice has yielded the smallest indices on all datasets. The overhead for the index can be roughly divided into two parts, i.e. control patterns and encoded data. Group-based algorithms will save space for control patterns by sharing the pattern in the group but take up more space for the encoded data due to the alignment requirement. Therefore, the final index size is determined by a trade-off between the above parts for Group-based algorithm. For example, our algorithms Group-Simple and Group-AFOR take up a bit more space than their corresponding sequential algorithms, i.e. Simple-9, and AFOR; while our algorithms GSC-1-CU (extended from Elias Gamma) and GSC-8-IU (extended from G8IU) result in much smaller index sizes compared to Elias Gamma and G8IU respectively.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have studied the problem of optimizing compression algorithms by using SIMD instruction sets. We generalize the ideas of previous studies for the incorporation of SIMD-based vectorization, and propose a general compression framework. Based on this framework, we develop several novel compression algorithms, including Group-Simple, Group-Scheme, Group-AFOR and Group-PFD, together with their corresponding vectorized implementations. We constructed extensive experiments on four large datasets and found that our algorithms performed consistently well on all the datasets. For example, the proposed SIMD-Group-Simple has very competitive query evaluation speed. The proposed Group-Scheme has the flexibility to tune on different compression metrics with two adjustable factors: the variants with larger compression granularity have more competitive encoding/decoding speed, and the variants with smaller compression granularity have more competitive compression ratio. We summarize the major highlights of our work in Table X.

Table IX. Summary of the highlights of our proposed algorithms.

Algorithm Categories	**Algorithms Based on our framework**	**Highlights**
Bit aligned	Group-Scheme	Flexible control over the trade-off between decoding speed and compression ratio with different compression granularity and length descriptor
Byte aligned		
Word aligned	Group-Simple	Very competitive query processing speed
Frame fixed bit-width	BP-128[Lemire and Boystov 2013] Group-PFor Group-AFOR	The fastest encoding/decoding speed (SIMD-BP128) and competitive encoding/decoding speed (SIMD-Group-PFD)

In the future, we will study how to apply SIMD techniques on other Intel architecture, i.e. 256-bit or the coming 512-bit vector registers with AVX instructions, and other architectures including PowerPC and ARM. Furthermore, we will try to test the algorithms on other domains, such as database and image processing.

REFERENCES

Anh V. N., and Moffat A. 2005. Inverted index compression using word-aligned binary codes. *Information Retrieval* 8.1 (2005): 151-166.

Anh V. N., and Moffat A. 2006. Improved word-aligned binary compression for text indexing. *Knowledge and Data Engineering, IEEE Transactions on* 18.6 (2006): 857-861.

Anh V. N., and Moffat A. 2010. Index compression using 64-bit words. *Software: Practice and Experience*

40.2 (2010): 131-147.

Broder A. Z., Carmel D., Herscovici M., Soffer, M., and Zien J. 2003. Efficient query evaluation using a two-level retrieval process. *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*. ACM, 426-434.

Büttcher S., Clarke C., and Cormack G. V. 2010. *Information retrieval: Implementing and evaluating search engines*. The MIT Press.

Chatterjee S., Bachega L. R., Bergner P., Kenneth A. D., Gunnels J. A., Gupta M., Gustavson F. G., Lapkowski C. A., Liu G. K., Mendell M. P., Nair R. D., Wait C. D., Ward C., and Wu P. 2005. Design and exploitation of a high-performance SIMD floating-point unit for Blue Gene/L. IBM Journal of Research and Development 49(2-3): 377-392 (2005)

Dean J. 2009. Challenges in building large-scale information retrieval systems: invited talk. *Proceedings of the Second ACM International Conference on Web Search and Data Mining*. ACM.

Ding S, and Suel T. 2011. Faster top-k document retrieval using block-max indexes. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM, 993-1002.

Delbru R., Campinas S., and Tummarello G. 2012. Searching web data: An entity retrieval and high-performance indexing model. *Web Semantics: Science, Services and Agents on the World Wide Web* 10 (2012): 33-58.

Elias P. 1975. Universal codeword sets and representations of the integers. *Information Theory, IEEE Transactions on* 21.2 (1975): 194-203.

Heman S. 2005. Super-scalar database compression between RAM and CPU-cache. *Master's thesis, Centrum voor Wiskunde en Informatica Amsterdam* (2005).

Intel Corporation. 2010. Intel 64 and IA-32 Architectures Software Developers Manual(Version 37). Santa Clara, California, USA, Intel Corporation, 2010.

Inkster D., Zukowski M., and Boncz P. 2011. Integration of VectorWise with Ingres. *SIGMOD Rec.* 40, 3 (November 2011), 45-53.

Jonassen S., and Bratsberg S. E. 2011. Efficient compressed inverted index skipping for disjunctive text-queries. *Advances in Information Retrieval*. Springer Berlin Heidelberg, 2011. 530-542.

Kwak H., Lee C., Park H., and Moon S. What is Twitter, a social network or a news media?. *Proceedings of the 19th International World Wide Web Conference*. ACM, 2010, 591-600.

Liu K., Qin X., Yan X., and Quan L. 2006. A SIMD Video Signal Processor with Efficient Data Organization. *Proceedings of IEEE Asian Solid-State Circuits Conference*, 115-118.

Lemke C., Sattler K. U., Faerber F., and Zeier A. 2010. Speeding up queries in column stores: a case for compression. *Proceedings of the 12th international conference on Data warehousing and knowledge discovery (DaWaK'10).*

Lomont C. 2011. Introduction to intel advanced vector extensions. *Proceedings of the 2nd Annual ASCI Conference*.

Lemire D., and Boytsov L. 2013. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*.

Ma W, and Yang C. 2002. Using Intel Streaming SIMD Extensions for 3D Geometry Processing. *Proceedings of IEEE Pacific Rim Conference on Multimedia*. 1080-1087.

Manning C. D., Raghavan P., and Schütze H. 2008. *Introduction to information retrieval*. Vol. 1. Cambridge: Cambridge University Press.

Navarro G,. De Moura E. S., Neubert M., Ziviani N., and Yates R. B. 2000. Adding compression to block addressing inverted indexes. *Information retrieval* 3.1 (2000): 49-77.

Rice R., and Plaunt J. 1971. Adaptive variable-length coding for efficient compression of spacecraft television data. *Communication Technology, IEEE Transactions on* 19.6 (1971): 889-897.

Robertson S. E., Walker S., Beaulieu M., and Willett P. 1999. Okapi at TREC-7: automatic ad hoc, filtering, VLC and interactive track. *Nist Special Publication SP* (1999): 253-264.

Raman V., Attaluri G., Barber R., Chainani N., Kalmuk D., KulandaiSamy V., Leenstra J., Lightstone S., Liu S., Lohman G. M., Malkemus T., Mueller R., Pandis I., Schiefer B., Sharpe D., Sidle R., Storm A., and Zhang L. 2013. DB2 with BLU acceleration: so much more than just a column store. *Proc. VLDB Endow. 6, 11 (August 2013)*, 1080-1091.

Scholer F., Williams H. E., Yiannis J., and Zobel, J. 2002. Compression of inverted indexes for fast query evaluation. *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 222-229.

Schlegel B., Gemulla R., and Lehner W. 2010. Fast integer compression using SIMD instructions. *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. ACM, 34-40.

Silvestri F., and Venturini R. VSEncoding. 2010. Efficient coding and fast decoding of integer lists via dynamic programming. *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*. ACM, 1219-1228.

Stepanov A. A., Gangolli A. R., Rose D. E., Ernst R. J., and Oberoi R. S. 2011. SIMD-based decoding of posting lists. *Proceedings of the 20th ACM international conference on Information and knowledge* *management*. ACM, 317-326.

Shan D., Ding S., He J., Yan H., and Li X. 2012. Optimized top-k processing with global page scores on block-max indexes. *Proceedings of the 5th ACM international conference on Web search and data mining*. ACM, 423-432.

Willhalm T., Popovici N., Boshmaf Y., Plattner H., Zeier A., and Schaffner J. 2009. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *Proc. VLDB Endow. 2, 1 (August 2009),* 385-394.

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

Walder J., Krátký M., Bača R., Platoš J., and Snášel V. 2012. Fast decoding algorithms for variable-lengths codes. *Information Sciences* 183.1 (2012): 66-91.

Yan H., Ding S., and Suel T. 2009. Inverted index compression and query processing with optimized document ordering. *Proceedings of the 18th International World Wide Web Conference*. ACM.

Zukowski M., Heman S., Nes N., and Boncz, P. 2006. Super-scalar RAM-CPU cache compression. *Proceedings of the 22nd International Conference on Data Engineering*. IEEE (2006): 59-71.

Zhang J., Long X., and Suel T. 2008. Performance of compressed inverted list caching in search engines. *Proceedings of the 17th International World Wide Web Conference*. ACM, 387-396.