SIMD-Based Decoding of Posting Lists

Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose
Ryan J. Ernst, Paramjit S. Oberoi

June 2014

Variable-length integer encodings are commonly used for storing posting lists in search engines. In this paper, we explore several variable-length integer encodings. Based on their common structure, we define a taxonomy for these encodings. In turn, this taxonomy suggests some novel encodings amenable to SIMD-based decoding. By applying appropriate SIMD instructions available on many general-purpose processors, we show significant decoding speedup compared to previously published techniques.

A9 Technical Report
A9TR-2011-01 rev. 2
Cleared for external distribution

1. Introduction

The central data structure in search engines is the *inverted index*, a mapping from index terms to the documents that contain them. The set of documents containing a given word is known as a *posting list*. Documents in posting lists are typically represented by unique nonnegative integer identifiers. Posting lists are generally kept in sorted order to enable fast set operations which are required for query processing.

Because posting lists typically account for a large fraction of storage used by the search engine, it is desirable to compress the lists. Smaller lists mean less memory usage, and in the case of disk-based indices, smaller lists reduce I/O and therefore provide faster access. A common way to compress posting lists is to replace document IDs in the list with differences between successive document IDs, known as Δ-*gaps* (sometimes just *gaps* or *d-gaps*). Since Δ-gaps are on average necessarily smaller than raw document IDs, we can use a variable-length unsigned integer encoding method in which smaller values occupy less space.

Since in a sorted list the Δ-gaps are always non-negative, we are only concerned with encoding nonnegative integers, and for the remainder of the paper "integer" means unsigned integer.

The integer encoding for a posting list is performed infrequently, at indexing time. The decoding, however, must be performed for every uncached query. For this reason, efficient integer decoding algorithms are essential, as are encoding formats that support such efficient decoding.

We started by exploring several commonly-used variable-length integer encodings and their associated decoding algorithms. Based on their common structure, we defined a taxonomy that encompassed these existing encodings and suggested some novel ones. Our investigation was motivated by the desire to incorporate fine-grained parallelism to speed up integer decoding. We were able to develop decoding methods that use SIMD instructions available on many general-purpose processors, in particular Intel and AMD processors. In this paper, we present these methods and evaluate them against traditional techniques. Our results indicate significant performance benefit from applying SIMD to these new formats.

The remainder of the paper is organized as follows. We review some related work on encoding formats and decoding algorithms in Section 2, and then present the taxonomy of byte-oriented encodings that we developed in Section 3. Section 4 describes three particular encoding formats which are well suited to SIMD parallelism in more detail. Section 5 gives an overview of our use of SIMD on Intel-compatible processors. In Section 6 we provide SIMD algorithms for decoding the formats introduced in Section 4. Section 7 contains an evaluation of the results, followed by our conclusions in Section 8.

2. Related Work

The problem of efficiently encoding integers has been studied for several decades. General compression techniques such as Huffman coding often utilize analysis of the data to choose optimal representations for each codeword, and generally allow arbitrary bit lengths. In information retrieval applications, however, the most common techniques for posting list compression use byte-oriented representations of integers and do not require knowledge of the data distribution.[1]

Perhaps the most common such format involves using 7 bits of each byte to represent data, and the 8th bit to represent whether or not the encoded representation of the integer continues into the next byte (see Figure 1). This format has a long history, dating back at least to the MIDI specification in the early 1980s (MIDI Manufacturers Association, 2001), and foreshadowed by earlier work on

[1]Büttcher et al. (2010a) refer to codes that do not depend on the data distribution as *nonparametric* codes.

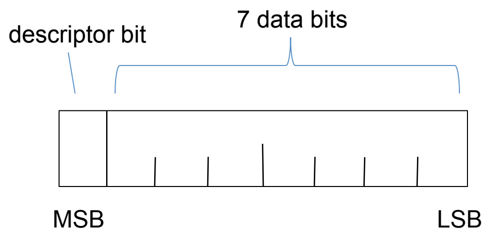


Figure 1: Format known in the literature as "vbyte", "vint", etc. Called *varint-SU* in the taxonomy of Section 3.

byte-based vocabulary compression by H.S. Heaps (1972). Cutting and Pedersen (1990) used this format to encode Δ-gaps in an inverted index. The information retrieval literature refers to the format by many different names, including vbyte, vint, and VB.

Another format *BA*, introduced by Grossman (1995) uses 2 bits of the first byte to indicate the length of the encoded integer in binary. In his WSDM keynote, Dean (2009) described a format he calls *group varint* that extends the BA format (which he calls *varint-30*) and reported significant performance gains obtained at Google by adopting it. A similar but more general format was described by Westmann et al. (2000) in the database context. Schlegel et al. (2010) applied SIMD-based decompression algorithms to a specialized version of Westmann's format that coincides exactly with group varint, but under the name *k-wise null suppression*. While they do not precisely describe the decoding and table generation, we believe that their algorithms are special cases of the generalized algorithms described in Section 6. All of these encodings fall into the taxonomy defined in this paper.

Anh (2004; 2005) introduced a family of *word-aligned* encodings with *Simple-9* being the fastest. Büttcher et al. (2010a) have compared performance of several encoding techniques using posting lists from the GOV2 corpus on a particular query sample. They originally reported vbyte being the fastest at decoding, with Simple-9 second. They recently updated their analysis (Büttcher et al., 2010b) to include the group varint format, reporting that it outperforms both vbyte and Simple-9. The SIMD techniques described here appear to do better. We discuss this further in Section 7.

3. Byte-Oriented Encodings

Encoding formats are generally distinguished by the granularity of their representation. We focus on encodings satisfying the following definition.

Definition 1 *We call an encoding* byte-oriented *if it satisfies the following conditions:*

1. *All significant bits of the natural binary representation are preserved.*
2. *Each byte contains bits from only one integer.*
3. *Data bits within a single byte of the encoding preserve the ordering they had in the original integer.*
4. *All bits from a single integer precede all bits from the next integer.*

A byte-oriented encoding is fixed-length *if every integer is encoded using the same number of bytes. Otherwise it is* variable-length.

Table 1: Nomenclature of Byte-Oriented Integer Encoding Formats

Descriptor Arrangement	Descriptor Length Encoding	Abbreviation	Names in the Literature
split	unary	varint-SU	v-byte (Croft et al., 2010), vbyte (Büttcher et al., 2010a), varint (Dean, 2009), VInt (Apache Software Foundation, 2004), VB (Manning et al., 2008)
packed	unary	varint-PU	none (proposed in this paper)
group	unary	varint-GU	none (proposed in this paper)
split	binary	varint-SB	none
packed	binary	varint-PB	BA (Grossman, 1995), varint30 (Dean, 2009)
group	binary	varint-GB	group varint (Dean, 2009), *k*-wise (*k*=4) null suppression (Schlegel et al., 2010)

Since variable-length byte-oriented formats must encode the length of the encoded data, they vary along the following three dimensions:

- The length can be expressed in binary or unary.
- The bits representing the length can be stored adjacent to the data bits of the corresponding integer so that some data bytes contain both data and length information; alternatively, lengths of several integers can be grouped together into one or more bytes distinct from the bytes containing data bits.
- If the length is represented in unary, the bits of the unary representation may be packed contiguously, or split across the bytes of the encoded integer.

It is evident that for byte-oriented formats, the natural unit of length is a byte. We call the set of bits used to represent the length the *descriptor*, since it describes how the data bits are organized.

We assume that each encoded integer requires at least one byte, so both binary and unary descriptors can represent the length n by recording the value $n - 1$. This reduces the number of bits required to represent a given length.[2]

The dimensions listed above provide the basis of a taxonomy of byte-oriented encoding formats for integers that can be encoded in four bytes or less. Selecting one of the possible options for each dimension determines a position in the taxonomy. This taxonomy provides a unifying nomenclature for the encoding formats, several of which have been described previously under various names. This taxonomy is shown in Table 1.

For example, Grossman's BA format becomes *varint-PB* in our taxonomy, since it is a *variable*-length encoding of *integers* with descriptor bits *packed* together and representing the length in *binary*.

For the unary formats, we follow the natural convention where the quantity is represented by the number of consecutive 1 bits, followed by a terminating 0. We start from the least significant bit. Thus 0111 represents the number 3.

Accordingly, the vbyte encoding may be viewed as representing the length $- 1$ of the encoded representation in the sequence of continuation bits. For example, a three-byte integer encoding would look like this:

[2]Storing length as n would allow the length zero to represent an arbitrary constant with zero data bytes. Such an encoding, however, does not in general satisfy the first property of Definition 1.

```
    1xxxxxxx
    1xxxxxxx
    0xxxxxxx
```

Notice that the leading bits form the unary number 2, representing the length 3. Thus we call this representation *varint-SU*, since it is a *variable*-length representation of *integers* with length information *split* across several bytes and represented in *unary*. While unary length representation has been widely used in bit-oriented encodings, for byte-oriented encodings the concept of continuation bits obscured their interpretation as unary lengths.

If binary length descriptors are used, the descriptor length must be fixed in advance, or additional metadata would be required to store the length of the descriptor itself. For this reason, all binary formats in the taxonomy use fixed-length descriptors of 2 bits per integer. Furthermore, since splitting a fixed-length k-bit binary descriptor (one bit per byte) results in a byte-oriented integer encoding that requires at least k bytes, the split binary encoding format does not offer a competitive compression rate and we do not consider it further.

There are also additional variations of some of these formats. Bytes of the encoded data may be stored in little-endian or big-endian order; descriptor bits may be stored in the least significant or most significant bits. While these choices are sometimes described as arbitrary conventions, in practice there are efficiency considerations that make certain variants attractive for certain machine architectures. For example, in varint-SU, representing termination as 0 in the most significant bit allows the common case of a one-byte integer to be decoded without any shifts or masks. While traditional decoding algorithms run more efficiently when the representation preserves native byte ordering, the performance of the SIMD algorithms presented in Section 6 does not depend on the ordering. Without loss of generality, for the remainder of the paper we restrict our attention to little-endian encodings.

The byte-oriented encoding taxonomy suggests two encodings, *varint-PU* and *varint-GU*, that, to our knowledge, have not been previously described.

Varint-PU is similar to varint-SU, but with the descriptor bits packed together in the low-order bits of the first byte rather than being split across all bytes. (The choice of low-order bits to hold the descriptor is appropriate for little-endian encodings on little-endian architectures so that all data bits for one integer are contiguous. For the same reason, on big-endian architectures placing the descriptor in the high-order bits and using big-endian encoding is more efficient to decode.) The compression rate of varint-PU is the same as that of varint-SU, since the bits in each encoded integer are identical but rearranged. The decoding performance of varint-PU using unaligned reads, masks, and shifts in a table-driven algorithm similar to that for varint-PB (see Dean, 2009, ”varint30”)] is faster than traditional varint-SU, but significantly slower than the group formats such as varint-GU, which is described in detail in the next section.

4. The Group Encoding Formats

Within our taxonomy, encoding formats that group several integers together provide opportunities for exploiting SIMD parallelism. These encodings satisfy the following important property.

Definition 2 *We call a byte-oriented encoding* **byte-preserving** *if each byte containing significant bits in the original (unencoded) integer appears without modification in the encoded form.*

Neither split or packed formats satisfy this property, since the descriptor bits are intermingled with data bits in some bytes. The separation of descriptor bytes from data bytes in group formats

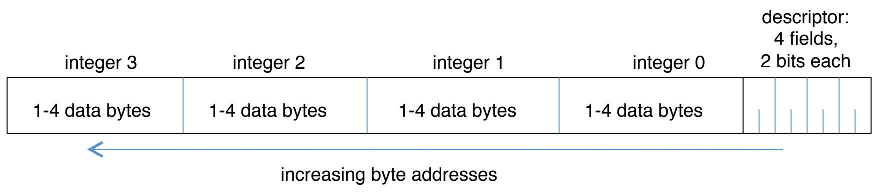


Figure 2: The varint-GB format.

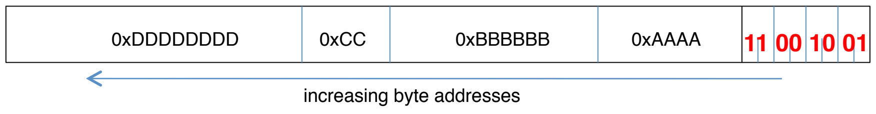


Figure 3: Storing the four integers `0xAAAA`, `0xBBBBBB`, `0xCC`, `0xDDDDDDDD` in varint-GB format. The value of each pair of bits in the descriptor is one less than the length of the corresponding integer. Byte addresses increase from right to left, matching the order of increasing bit significance. The order of pairs of bits in the descriptor matches the order of the integers.

allows for more efficient decoding. It facilitates the use of tables to simplify the decoding process and avoids bitwise manipulations that are required to eliminate interspersed descriptor bits. In particular, we shall see in Section 6 that byte-preserving encodings are especially amenable to decoding with the SIMD techniques described in this paper.[3]

There are two classes of group formats, group binary (varint-GB) and group unary (varint-GU).

In the *varint-GB* format (called group varint in (Dean, 2009) and k-wise null supression (with $k = 4$) in (Schlegel et al., 2010)) a group of four integers is preceded by a descriptor byte containing four 2-bit binary numbers representing the lengths of the corresponding integers. Figure 2 illustrates this format for one such group. The actual number of bytes in a group may vary from 4 to 16. Figure 3 shows how the four hexadecimal numbers `0xAAAA`, `0xBBBBBB`, `0xCC`, `0xDDDDDDDD` would be represented. The four integers require, correspondingly, 2 bytes, 3 bytes, 1 byte, and 4 bytes. For each integer, its length n is represented in the descriptor by the 2-bit binary value $n - 1$. Therefore, the descriptor byte contains the values 01, 10, 00, and 11 respectively. To maintain a consistent order between descriptor bits and data bytes, we store the first binary length in the least significant bits, and so on. Thus the descriptor byte for these four integers is 11001001.

Varint-GB operates on a fixed number of integers occupying a variable number of bytes, storing their lengths in binary. In contrast, the varint-GU format operates on a fixed number of bytes encoding a variable number of integers, storing their lengths in unary.

Varint-GU groups 8 data bytes together along with one descriptor byte containing the unary representations of the lengths of each encoded integer. The 8 data bytes may encode as few as 2 and as many as 8 integers, depending on their size. The number of zeros in the descriptor indicates the number of integers encoded. This format is shown in Figure 4. The block size of 8 is the minimal size that can use every bit of the descriptor byte; larger multiples of 8 are possible, but did not improve performance in our experiments.

Since not every group of encoded integers fits evenly into an 8-byte block, we have two variations of the encoding: incomplete and complete.

[3]Note that what Anh (2004) calls *word-aligned* is neither byte-oriented nor byte-preserving as defined in this paper.

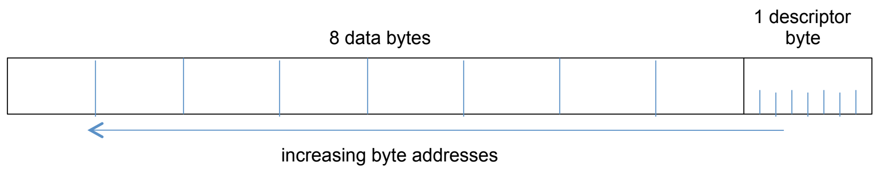


Figure 4: The varint-GU format.

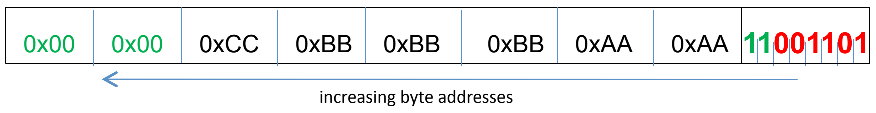


Figure 5: Storing the three integers (`0xAAAA`, `0xBBBBBB`, `0xCC`) in varint-G8IU format. The descriptor bits express the unary lengths of the integers. Since the next integer `0xDDDDDDDD` in our example does not fit in the data block, the block is left incomplete and padded with 0s, while the descriptor is padded with 1s.

In the *incomplete* block variation, which we call *varint-G8IU*, we store only as many integers as fit in 8 bytes, leaving the data block incomplete if necessary.[4] The remaining space is padded with zeros, but is ignored on decoding.[5] When there is no additional integer to decode, the final (most significant) bits of the descriptor will be an unterminated sequence of 1 bits.

An example is shown in Figure 5. We use the same four integers `0xAAAA`, `0xBBBBBB`, `0xCC` and `0xDDDDDDDD` to illustrate. Encoding these values requires 10 bytes, but we have only 8 bytes in the block. The first three integers fit into the block using 6 bytes, leaving 2 bytes of padding. The final integer `0xDDDDDDDD` is left for the next block (not shown). The descriptor contains the three unary values 01, 011, and 0, and two padding bits 11. These are arranged in the same order as the integers, giving the descriptor a binary value of 11001101.

In the *complete* block variation, which we call *varint-G8CU*, we always fill all eight bytes in a data block.[6] As before, the number of zero bits in the descriptor indicates the number of complete integers encoded. In situations where an integer exceeds the remaining space in the current block, as much of that integer as fits is placed in the current block. The remaining bytes of that integer are carried over into the next data block. Similarly, the corresponding descriptor bits are carried over to the next block's descriptor byte.

An example is shown in Figure 6. Again we use the same four integers `0xAAAA`, `0xBBBBBB`, `0xCC` and `0xDDDDDDDD`. The first three integers and the corresponding descriptor bits are stored exactly as in varint-G8IU. However, varint-G8CU handles the fourth integer differently. Its first two bytes are placed in the first data block, filling it entirely, and the remaining two bytes go into the following block. The two descriptor bits corresponding to these last two bytes go into the next block's descriptor byte. Although spread across two descriptor bytes, the unary value of the descriptor bits for this fourth integer still represents the length $-$ 1 of the encoded integer.

[4]In our notations for the encoding, the number 8 represents the size of the data block.

[5]There is also a variation of this encoding format that uses variable size data blocks and avoids padding. Its performance characteristics are between those of varint-G8IU and varint-G8CU described later.

[6]As before, the number 8 represents the size of the data block.

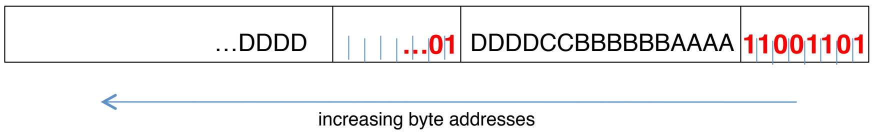


Figure 6: Storing four integers (`0xAAAA`, `0xBBBBBB`, `0xCC` and `0xDDDDDDDD`) in varint-G8CU format. The last two bytes of the fourth integer carry over to the subsequent data block, and its descriptor bits carry over to the subsequent descriptor byte.

5. SIMD on Intel/AMD

Facilities for fine-grained parallelism in the SIMD (Single Instruction, Multiple Data) paradigm are widely available on modern processors. They were originally introduced into general-purpose processors to provide vector processing capability for multimedia and graphics applications. Although SIMD instructions are available on multiple platforms, we restricted our focus to Intel-compatible architectures (Intel Corporation, 2010) implemented in processors by Intel and AMD and in extensive use in many data centers.

In these architectures, a series of SIMD enhancements have been added over time. Among the current SIMD capabilities are 16-byte XMM vector registers and parallel instructions for operating on them.

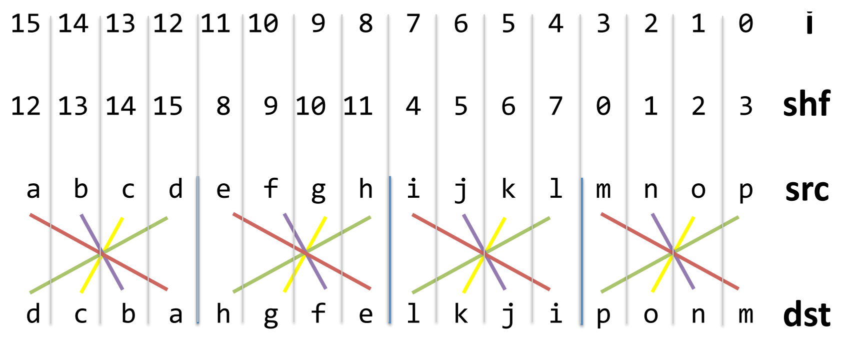
The `PSHUFB` instruction, introduced with SSSE3 in 2006, is particularly useful.[7] It performs a permutation ("shuffle") of bytes in an XMM register, allowing the insertion of zeros in specified positions. `PSHUFB` has two operands, a location containing the data and a register containing a shuffle sequence. If we preserve the original value of the data operand, we can view `PSHUFB` as transforming a source sequence of bytes *src* to a destination sequence *dst* according to the shuffle sequence *shf*.

```
for $0 \le i < 16$ parallel do
    if $shf[i] \ge 0$ then
        $dst[i] \leftarrow src[shf[i] \bmod 16]$
    else
        $dst[i] \leftarrow 0$
end
```

In other words, the *i*th value in the shuffle sequence indicates which source byte to place in the *i*th destination byte. If the *i*th value in the shuffle sequence is negative, a zero is placed in the corresponding destination byte.

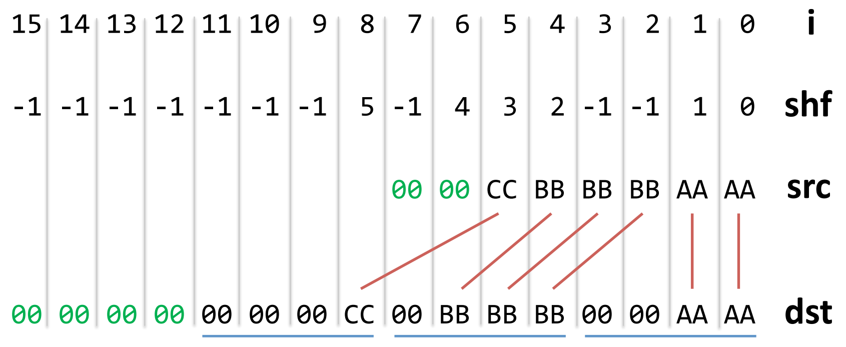
The example illustrated in Figure 7 shows how `PSHUFB` can be used to reverse the byte order of four 32-bit integers at once.

[7] A similar instruction, `vperm`, was introduced in the AltiVec/VMX instruction set for the PowerPC processor developed between 1996 and 1998 and is part of the standard Power Instruction Set Architecture (Power.org, 2010). While it operates differently, it can also be used to provide essentially the same functionality for the purposes of the algorithms described in this paper.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 i
12 13 14 15 8 9 10 11 4 5 6 7 0 1 2 3 shf
a b c d e f g h i j k l m n o p src
d c b a h g f e l k j i p o n m dst

Figure 7: Using the PSHUFB instruction to reverse the byte order of four integers in parallel. The shf vector determines the shuffle sequence used to transform src to dst.



15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 i
-1 -1 -1 -1 -1 -1 -1 5 -1 4 3 2 -1 -1 1 0 shf
00 00 CC BB BB BB AA AA src
00 00 00 00 00 00 00 CC 00 BB BB BB 00 00 AA AA dst

Figure 8: Using the PSHUFB instruction to decode the varint-G8IU format.

6. SIMD Decoding

Since byte-preserving formats remove leading zero bytes while retaining the significant bytes intact, the essence of decoding is reinserting the zero bytes in the right places. We construct a shuffle sequence by inserting -1s in a sequence $\{0, 1, 2, 3, ...\}$. With this sequence, the PSHUFB instruction will copy the significant data bytes while inserting the missing zeros.

An example of using PSHUFB to decode varint-G8IU is shown in Figure 8. This is the same data represented in Figure 5.

For a given format, we can precompute what the correct shuffle sequence is for a particular data block and its corresponding descriptor byte. For all possible values of the descriptor (and sometimes additional state) we build a table of any shuffle sequence that might be needed at decode time.

The table entries also contain a precomputed offset. For the varint-GB format, the offset indicates how many bytes were consumed to decode 4 integers; it always outputs 16 bytes. For the varint-GU formats, the offset indicates how many integers were decoded; it always consumes 8 bytes.

Table construction occurs only once, while table lookup occurs every time a group is decoded.

Given the availability of these tables, the general strategy of all the decodings is:

1. read a chunk of data and its corresponding descriptor;

2. look up the appropriate shuffle sequence and offset from the table;
3. perform the shuffle;
4. write the result;
5. advance the input and output pointers.

The rest of the section describes the implementations of each step for all three formats.

This approach allows us to decode several integers simultaneously with very few instructions. It requires no conditionals, and thus avoids performance penalties due to branch misprediction. Two techniques make this possible. First, the logical complexity has been shifted from the code to the table. Second, the algorithm always reads and writes a fixed amount and then relies on the table to determine how much input data or output data it has actually processed.[8]

Data blocks are not aligned on any fixed boundary. We depend on the ability of the CPU to perform unaligned reads and writes efficiently, and we have observed this to be true on modern Intel processors.

6.1 Details of the Decoding Algorithm

All of the group formats can be decoded using the generalized algorithm shown in Algorithm 1.

Algorithm 1: decodeBlock

```
Decodes a block of data using SIMD shuffle.
input  : $src, dst, state$
output : $src, dst, state$
begin
    $data \leftarrow$ read($src + 1, 16$)
    $entry \leftarrow table_{format}[desc, state]$
    $shf \leftarrow$ shuffleSequence($entry$)
    shuffleAndWrite($data, shf, dst$)
    $src \leftarrow src +$ inputOffset($entry$)
    $dst \leftarrow dst +$ outputOffset($entry$)
    return $src, dst$, nextState($entry$)
end
```

Because this algorithm constitutes the inner loop of the decoding process, it is essential to inline the implementation to avoid function call overhead. The algorithm takes three inputs:

- *src* – a pointer to the input byte stream of encoded values
- *dst* – a pointer to the output stream of integers in case of varint-GB, and varint-G8IU; in the case of varint-G8CU, *dst* is a pointer to an output stream of bytes, since decoding a block of varint-G8CU may result in writing a portion of a decoded integer.
- *state* – auxiliary state. This is required only for varint-G8CU, where it is an integer i, with $0 \leq i < 4$ indicating the number of bytes modulo 4 of the last integer written.

[8]This requires that the input and output buffers always have at least this amount available to read or write.

It reads encoded values from the input stream, outputs decoded integers to the output stream and returns as its result the new positions of *src*, *dst*, and the updated *state*.

We always read 16 bytes, which is the size of the XMM register used by the `PSHUFB` operation. The number of bytes corresponding to a single byte descriptor is 8 for the unary formats and at most 16 for the binary format. While it is possible to read only 8 bytes for the unary formats, in our evaluation we found that doing so did not improve performance, and in fact made the implementation slightly slower.

A different table is used for each format. There is a table entry corresponding to each possible descriptor value and state value. The table has 256 entries for the varint-GB and varint-G8IU formats. For varint-G8CU format, the table has $4 \times 256 = 1024$ entries, because we have an entry for each descriptor and state pair, and the state is an integer i, with $0 \leq i < 4$.

Each table entry logically contains four things:

- a shuffle sequence
- an input offset
- an output offset
- the state value to use for the subsequent block

The shuffleSequence, inputOffset, outputOffset, and nextState functions are accessors for these fields. For some of the formats, some of these values are constant over all entries in the table, and are not stored explicitly; the accessors simply return constant values.

The shuffleAndWrite operation uses the `PSHUFB` operation with the provided shuffle sequence to expand the 16 bytes of data inserting zeros into the correct positions, and then writes its result to the destination.

In the varint-GB case, the shuffle sequence is a 16-byte sequence describing a single `PSHUFB` operation. A single `PSHUFB` is sufficient because the group always contains four encoded integers, and thus the output never exceeds 16 bytes.

For decoding the varint-GU formats, the shuffle sequence is a 32-byte sequence specifying two `PSHUFB` operations. The second `PSHUFB` is required for the unary formats because an 8-byte data block may encode up to 8 integers, which can expand to 32 bytes. The output of the first `PSHUFB` is written to locations beginning at *dst*, and the output of the second `PSHUFB` to locations beginning at $dst + 16$. To avoid conditionals, the second shuffle is always performed, even when the output does not exceed 16 bytes. Since `PSHUFB` rearranges the register in place, the corresponding register needs to be reloaded with the original data before the second `PSHUFB`.

For unary formats, the input offset, by which we increment the *src*, is always 8 bytes. For varint-G8IU, the output offset measured in units of decoded integers varies between 2 and 8, except for the last block of a sequence, which may contain only 1 integer. For varint-G8CU, decoding one block may result in writing a portion of a decoded integer, so the output is a byte stream and the offset is measured in byte units. It varies between 8 and 32 bytes, except for the last block of the sequence which may output only 1 byte.

In the case of varint-GB, the output offset is always a constant 4 integers.[9] The input offset varies between 4 and 16 bytes.

For all of the encodings, the input offset needs to account for the additional one byte of the descriptor as well. All variable offsets are precomputed and stored in the format table.

[9]The varint-GB format requires auxiliary information to deal with sequences of length not divisible by 4. This may be done using length information stored separately or the convention that zero values do not appear in the sequence, so terminal zeros can be ignored.

For the varint-G8CU format, the table also contains the new state information indicating the number of bytes in the last integer to be used to decode the subsequent block.

6.2 Building the Tables

For each of the group formats, the decoding table used by Algorithm 1 is constructed in advance. The construction process takes as input a descriptor byte value and a state value. It builds the shuffle sequence for the entry and computes the input offset, output offset, and next state (unless they are constant for the format) .

We assume we deal only with valid descriptor values, those which could actually arise from encoding. For varint-GB, all possible byte values are valid. For the group unary formats, a descriptor is valid if and only if the distance between consecutive zero bits does not exceed 4.

The algorithms for constructing shuffle sequences, offset values, and the next state value depend on the following abstract functions:

- $\mathrm{num}(desc)$ gives the number of integers whose encoding is completed in the group described by the descriptor value *desc*. For varint-GB this is always 4. For the group unary formats, this value is the number of 0 (termination) bits in *desc*.
- $\mathrm{len}(desc, i)$ gives the length of the *i*th integer in the group, for each i, $0 \le i < \mathrm{num}(desc)$. This is the length determined by the *i*th individual bit pair in *desc* for varint-GB, or the *i*th unary value in *desc* for the unary formats.
- $\mathrm{rem}(desc)$ gives the number of bytes modulo 4 in the last encoded integer in the group. This is needed only for varint-G8CU, where it is equal to the number of leading 1s in the descriptor *desc*. For the other formats it is always 0.

Again, the basic idea in constructing a shuffle sequence is to insert -1s in a sequence {0, 1, 2, 3, ...} representing the byte positions in one block of the source data being decoded. The resulting shuffle sequence is used by the `PSHUFB` instruction to copy the significant data bytes while inserting the missing leading zeros. The details of the construction are shown in Algorithm 2. The algorithm takes two inputs:

- *desc* the descriptor value
- *state* the number of bytes modulo 4 written from the last integer in the prior group. For varint-GB and varint-G8IU, the value of state is always 0, since only complete integers are written in a given data block in these formats.

and produces one output, *shf*, the shuffle sequence to be used for the given descriptor and state. The first loop iterates over every completed integer in the group corresponding to the given descriptor. For each completed integer in the group, the inner loop sets the shuffle sequence to move the encoded bytes from the source of the shuffle operation, inserting -1s to produce the leading zeros necessary to complete the decoded integer. Here the variable j advances over the source data positions in the data block, while the variable k advances over the positions in the shuffle sequence, which correspond to destination positions of the shuffle operation.

The concluding loop only executes for varint-G8CU. It sets the remainder of the shuffle sequence to transfer encoded bytes from the source for the last incomplete integer in the group.

Computing input offsets is easy. For the unary formats the input offset is always 9; we always consume a block of 8 bytes of data and 1 descriptor byte. For the group binary format varint-GB,

Algorithm 2: constructShuffleSequence

```
input  : $desc, state$
output: $shf$
begin
    $j, k \leftarrow 0$
    $s \leftarrow 4 - state$
    for $0 \leq i < \mathrm{num}(desc)$ do
        for $0 \leq n < s$ do
            if n $< \mathrm{len}(desc, i)$ then
                $shf[k] \leftarrow j$
                $j \leftarrow j + 1$
            else
                $shf[k] \leftarrow -1$
            end
            $k \leftarrow k + 1$
        end
        $s \leftarrow 4$
    end
    for $0 \leq n < \mathrm{rem}(desc)$ do
        $shf[k] \leftarrow j$
        $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    end
    return $shf$
end
```

the input offset for a given descriptor *desc* is

$$1 + \sum_{i=0}^{3} \mathrm{len}(desc, i)$$

which is the sum of the lengths of the integers in the group plus 1 for the descriptor byte.

The output offset for varint-GB and varint-G8IU is equal to $\mathrm{num}(desc)$ integers (which is always 4 for varint-GB). The output offset is

$$4 \cdot \mathrm{num}(desc) - state + \mathrm{rem}(desc)$$

for the varint-G8CU format.

The state value for the subsequent block is always 0 for varint-GB and varint-G8IU (and the state can be ignored for these formats). For varint-G8CU it is $\mathrm{rem}(desc)$.

6.3 Applying SIMD to varint-SU

Although varint-SU is not a byte-preserving encoding, Algorithm 1 can be applied as a component for decoding it. By efficiently gathering the descriptor bits from each byte into a single value for a table lookup, we can treat a sequence of 8 consecutive varint-SU-encoded bytes almost as if they were a varint-GU-encoded block. The Intel instruction `PMOVMSKB`, which gathers the most-significant bits from 8 bytes into a single byte, provides the needed functionality. After applying a shuffle as in Algorithm 1, the descriptor bits must be "squeezed out" to finish the decoding; this removal of the interspersed descriptor bits requires masks and shifts. Despite this additional step, the SIMD implementation still outperforms the traditional method as shown in Section 7.

7. Evaluation

We used three corpora in our evaluation: Two from Amazon's U.S. product search service, and one from Wikipedia (Wikimedia Foundation, 2010). The first Amazon index, SI, contains general product information. The second Amazon index, STRIP, contains the full text of books available through the "Search Inside the Book" feature.

The C++ implementation was compiled using gcc 4.5.1.[10] Measurements were done using a single-threaded process on an Intel Xeon X5680 processor (3.3GHz, 6 cores, 12 MB Intel Smart Cache shared across all cores), with 12 GB DDR3 1333 MHz RAM. Measurements are done with all of the input and output data in main memory.

Decoding speed results are shown in Table 2. For each corpus, this table shows the decoding speed measured in millions of integers per second. The "traditional" implementation of varint-SU shown in the table is our best implementation for this encoding using traditional techniques. The "mask table" implementation of varint-GB is our implementation of the technique described in Dean (2009).

There seems to be no standard benchmark for measuring integer decoding implementations in the information retrieval field. Even with conventional test corpora, there are many variations possible in producing the posting lists. Evaluation methods are also not standardized. Some research on compression only reports compression rate but not speed. Those who report speed use different metrics and data.

[10]It uses gcc intrinsics `__builtin_ia32_pshufb128`, `__builtin_ia32_loaddqu`, and `__builtin_ia32_storedqu` to invoke the `PSHUFB` and unaligned load and store instructions required.

Table 2: Decoding rates (in millions of integers per second); larger is better

encoding	algorithm	Wikipedia	SI	STRIP
varint-SU	traditional	296	653	350
varint-SU	SIMD	540	692	568
varint-GB	mask table	847	846	844
varint-GB	SIMD	**1283**	1333	1272
varint-G8IU	SIMD	1272	**1608**	**1415**
varint-G8CU	SIMD	1097	1544	1286

Table 3: Compression ratios; smaller is better

encoding	Wikipedia	SI	STRIP
varint-SU	0.43	0.28	0.38
varint-GB	0.46	0.33	0.41
varint-G8IU	0.47	0.31	0.41
varint-G8CU	0.45	0.31	0.39

Büttcher et al. (2010a) start with the GOV2 corpus and the 10000 queries from the efficiency task of the TREC Terabyte Track 2006. They indicate they used components of the Wumpus search engine to index the GOV2 corpus, and then decoded the posting lists for non-stopword terms contained in the query set. They do in-memory measurements, but also compute a "cumulative overhead" for decoding and disk I/O by estimating the I/O time based on compression rate.

Schlegel et al. (2010) use 32MB synthetic data sets constructed by sampling positive 32-bit integers from the Zipf distribution with different parameter values. They measure the amount of uncompressed data that can be processed per second.

Dean (2009) uses millions of integers decoded per second as a performance metric (as we do), but he does not provide details on the data or evaluation method used in his measurements.

Creation of a standard benchmark containing several sequences of integers with distinct but representative statistical characteristics would allow meaningful comparisons of different implementations.

Compression ratios are shown in Table 3. Here the compression ratio indicates the ratio between the bytes required for the integers encoded in the format and their original size of 4 bytes each. Compression ratios depend only on the encoding and not on the implementation.

8. Conclusions and Future Work

We discovered a taxonomy for variable-length integer encoding formats. This led us to identify some new encodings that offer advantages over previously known formats. We identified the *byte-preserving* property of encoding formats which makes them particularly amenable to parallel decoding with SIMD instructions. The SIMD-based algorithms that we developed outperformed the traditional methods by a factor of 3 to 4, and the best known methods by over 50%. Furthermore the new group unary formats offer better compression than the group binary format on all of the corpora tested.

Since Schlegel et al. (2010) also reported success applying SIMD to Elias γ (Elias, 1975), which is not aligned on byte boundaries at all, further investigations are needed to see whether SIMD techniques can be applied to Simple-9 (Anh and Moffat, 2005).

We restricted our investigation to integers that can be encoded in four bytes or less. We believe, however, that some of the encodings that we introduced could be easily extended to larger values.

Further investigation is also needed to see how these encoding formats integrate into set operations, such as intersection, which are central to query processing.

SIMD instructions, a very powerful resource that is at present under-utilized, offer the opportunity for significant performance improvements in modern search engines.

Acknowledgments

We thank Thomas London for his helpful suggestions on an earlier draft of this paper.

Contact Information

The authors may be reached by e-mail at:

`stepanov@a9.com`, `gangolli@a9.com`, `rose@ordinology.com`,
`ryan@iernst.net`, `paramjit@a9.com`

or by post to:

A9.com
130 Lytton Ave.
Palo Alto, CA 94303

Appendix: C++ Code

In the C++ code below, we omit inline directives and we use `uint8` and `int8` to mean `uint8_t` and `int8_t`, respectively.

A. Generic Algorithms

A.1 GroupFormat

Our algorithms for decoding and for constructing shuffle sequences are defined for a set of types satisfying certain requirements, which we call GroupFormat.[11] A type T satisfies the requirements of GroupFormat if:

T contains the following nested types:

`shuffle_sequence_type`
: encapsulates a pointer to the shuffle sequence. It has a static constant member `shuffle_size` of type `size_t` whose value is the size in bytes of the shuffle sequence, 16 for a single shuffle, 32 for two shuffles. This constant should be made visible as a static constant member `shuffle_size` defined within GroupFormat itself.

`descriptor_type`
: defines the type of the descriptor. This definition is needed because of C++ language requirements. The type must be `uint8` for all GroupFormat types.

`info_type`
: encapsulates the input offset, the output offset, and the next state.

`state_type`
: encapsulates information about partially completed work after decoding the previous block. For the varint-G8CU format, this type is defined as `uint8`, and a value of the type gives the number of bytes in the last partially decoded integer. For the other formats, there is never partially completed work; therefore we use the type `empty_type`, described later.

T provides the following member functions. The first group of interfaces is used during decoding:

`lookup_shuffle(descriptor_type, state_type)` → `shuffle_sequence_type`
: given descriptor and state values, returns the shuffle sequence appropriate for that descriptor and state.

`lookup_info(descriptor_type, state_type)` → `info_type`
: given descriptor and state values, returns an instance of `info_type` that provides the right input offset, output offset, and next state for that descriptor and state.

`input_offset(info_type)` → `ptrdiff_t`
: given an instance of the `info_type`, returns the input offset determined by that instance.

`output_offset(info_type)` → `ptrdiff_t`
: given an instance of the `info_type`, returns the output offset determined by that instance.

`next_state(info_type)` → `state_type`
: given an instance of the `info_type`, returns the next state determined by that instance.

[11]Such a set of requirements on types is called a *concept*. For a formal treatment, see Stepanov and McJones (2009).

The second group of interfaces is used while building the table:

`valid_descriptor(descriptor_type)` $\to$ `bool`
: returns `true` if and only if the given descriptor is valid (could be seen while decoding a properly encoded sequence).

`num(descriptor_type)` $\to$ `uint8`
: (as defined in Section 6.2) gives the number of integers whose encoding is completed in the group described by the given descriptor.

`len(descriptor_type desc, uint8 state)` $\to$ `uint8`
: (as defined in Section 6.2) gives the length of the `i`th integer in the group, for each `i`, $0 \leq$ `i` $<$ `num(desc)`. This is the length determined by the *i*th individual bit pair in `desc` for varint-GB, or the `i`th unary value in `desc` for the unary formats. as defined in Section 6.2.

`rem(descriptor_type desc)` $\to$ `uint8`
: (as defined in Section 6.2) gives the number of bytes modulo 4 in the last encoded integer in the group. This is needed only for varint-G8CU, where it is equal to the number of leading 1s in the descriptor `desc`. For the other formats it is always 0.

A.2 Decoding One Block

Using the GroupFormat concept, we implement Algorithm 1 in C++.

Description:

Decodes an input block of a group format and writes the decoded integers

Arguments:

src A reference to a pointer to the input. The pointer should point to the descriptor byte, followed by a block of data. The pointer is incremented to point just after the decoded block.

dst A reference to a pointer to the output. Decoded integers will be written starting at this point. The pointer is incremented to point just after the last byte written.

state A reference to the state left after the prior call to `decode_block`. Before the first call, it should be set to the state `GroupFormat::state_type(0)`.

Code:

```
template<typename GroupFormat>
struct decode_block : GroupFormat {
    void operator ()(const uint8*& src, uint8*& dst,
                     typename GroupFormat::state_type& state) {
        auto desc = *src;
        auto shf = lookup_shuffle(desc, state);
        shuffle_and_store(src + 1, shf, dst);
        auto info = lookup_info(desc, state);
        src += input_offset(info);
        dst += output_offset(info);
        state = next_state(info);
    }
};
```

This defines a function object that extends the template type GroupFormat. Here the inheritance is unusual in that the base class is used to bring in the format-specific details, while the code in the subclass is generic. This structure also allows us to invoke the format-specific methods without qualifying their names.

The `auto` keyword, a facility introduced in the C++0x standard, creates a variable of the type returned by the initializing expression.

A.3 Shuffle Types and Functions

The `shuffle_and_store` function used in the preceding algorithm has two variant implementations, one for the 16-byte single-shuffle case and one for the 32-byte two-shuffle case. The proper variant is invoked based on the specific type of the shuffle sequence `shf` returned by the `lookup_shuffle` function.

The `shuffle` struct is parameterized by an integer specifying the size of the shuffle sequence in bytes. The `shuffle_sequence_type` of a GroupFormat always returns an instance of this template class.

```
template<size_t n>
struct shuffle {
    static const size_t shuffle_size = n;
    const aligned16byte_t* ptr;
    shuffle(const aligned16byte_t* p) : ptr(p) {}
};
```

The functions below implement `shuffle_and_store` for `shuffle` of size 16 and 32. The `shuffle_and_store128` function invokes the `PSHUFB` operation. It is described in Section D.

Description:

Applies `PSHUFB` once for a 16-byte shuffle or twice for a 32-byte shuffle

Arguments:

src A pointer to a block of input data of 16 bytes.

shf A shuffle sequence of either 16 bytes or 32 bytes.

dst A pointer to the output.

Code:

```
void shuffle_and_store(const uint8* src, shuffle<16> shf,
                       uint8* dst) {
    aligned16byte_t r = load_unaligned128(src);
    shuffle_and_store128(r, *shf.ptr, dst);
}

void shuffle_and_store(const uint8* src, shuffle<32> shf,
                       uint8* dst) {
    aligned16byte_t r = load_unaligned128(src);
    shuffle_and_store128(r, *shf.ptr, dst);
    shuffle_and_store128(r, *(shf.ptr + 1), dst + 16);
}
```

A.4 Construction of the Shuffle Sequence

This implements Algorithm 2 in C++ using the GroupFormat concept.

Description:

Constructs a shuffle sequence for a given descriptor and state

Arguments:

desc the 8-bit descriptor for which the shuffle sequence is to be constructed

state the state value for which the shuffle sequence is to be constructed

shf a pointer to the shuffle sequence to be written

Code:

```
template<typename GroupFormat>
struct construct_shuffle_sequence : GroupFormat {
    void operator()(typename GroupFormat::descriptor_type desc,
                    uint8 state, int8* shf) {
        std::fill_n(shf, GroupFormat::shuffle_size, -1);

        if (!valid_table_entry(desc, state)) return;

        uint8 j = 0;
        uint8 s = 4 - state;
        for (uint8 i = 0; i < num(desc); ++i) {
            for (uint8 n = 0; n < s; ++n, ++shf) {
                if (n < len(desc, i)) *shf = j++;
            }
            s = 4;
        }

        for (uint8 n = 0; n < rem(desc); ++n) {
            *shf++ = j++;
        }
    }
};
```

Here again, as was done for Algorithm 1 earlier, the algorithm is implemented as a function object extending the template parameter GroupFormat.

A.5 Auxiliary Components

A.5.1 The `cast_to_shuffle` function

The `cast_to_shuffle_builder` function is used by the constructor of a specific GroupFormat to simplify the call to `construct_shuffle_sequence`. It allows the constructed object to view itself as an instance of its subclass.

```
template<typename GroupFormat>
construct_shuffle_sequence<GroupFormat>&
cast_to_shuffle_builder(GroupFormat* f) {
    return static_cast<construct_shuffle_sequence<GroupFormat>&>(*f);
}
```

A.5.2 The empty type

The varint-GB and varint-G8IU format classes in the next section use `empty_type` as their `state_type` since no actual state is needed for those formats:

```
struct empty_type {
    empty_type() {}
    template<typename T> explicit empty_type(const T&) {}
    empty_type& operator=(const empty_type&) {
        return *this;
    }
};
```

B. Format Classes

This section contains definitions of the format-specific classes that meet the requirements of the GroupFormat concept.

B.1 varint-GB

The `varint_gb_format` class describes the varint-GB encoding as a GroupFormat.

```
struct varint_gb_format {

    typedef shuffle<16> shuffle_sequence_type;
    typedef uint8 descriptor_type;
    typedef uint8 info_type;
    typedef empty_type state_type;

    aligned16byte_t shuffles[256];
    uint8 input_offsets[256];

    // The first group of interfaces is used for decoding:

    shuffle_sequence_type lookup_shuffle(descriptor_type desc,
                                         state_type state) {
        return shuffle_sequence_type(&shuffles[desc]);
    }

    uint8 lookup_info(descriptor_type desc, state_type) {
        return input_offsets[desc];
    }
```

```
        ptrdiff_t input_offset(info_type r) {
            return ptrdiff_t(r);
        }

        ptrdiff_t output_offset(info_type) {
            return 16;
        }

        state_type next_state(info_type) {
            return state_type();
        }

        // The second group of interfaces is used for building the table:

        static const size_t shuffle_size = shuffle_sequence_type::shuffle_size;

        bool valid_table_entry(descriptor_type, uint8 state_value) {
            return state_value == 0;
        }

        uint8 num(descriptor_type desc) {
            return 4;
        }

        uint8 len(descriptor_type desc, uint8 i) {
            return ((desc >> (2 * i)) & 0x03) + 1;
        }

        uint8 rem(descriptor_type desc) {
            return 0;
        }

        uint8 offset(descriptor_type d) {
            // Sum of the four lengths + 1 for the descriptor
            return len(d, 0) + len(d, 1) + len(d, 2) + len(d, 3) + 1;
        }

        varint_gb_format() {
            auto shuffle_builder = cast_to_shuffle_builder(this);
            for(int i = 0; i < 256; ++i) {
                input_offsets[i] = offset(descriptor_type(i));
                shuffle_builder(descriptor_type(i), 0, (int8*)(shuffles + i));
            }
        }

    };
```

B.2 varint-G8IU

The varint_g8iu_format class describes the varint-G8IU encoding as a GroupFormat.

```
struct varint_g8iu_format {

    typedef shuffle<32> shuffle_sequence_type;
    typedef uint8 descriptor_type;
    typedef uint8 info_type;
    typedef empty_type state_type;

    aligned16byte_t shuffles[256][2];
    uint8 output_offsets[256];

    // The first group of interfaces are used for decoding:

    shuffle_sequence_type lookup_shuffle(descriptor_type desc,
                                         state_type state) {
        return shuffle_sequence_type(shuffles[desc]);
    }

    uint8 lookup_info(descriptor_type desc, state_type) {
        return output_offsets[desc];
    }

    ptrdiff_t input_offset(info_type) {
        return 9;  // size of data block + 1 for descriptor
    }

    ptrdiff_t output_offset(info_type r) {
        return ptrdiff_t(r);
    }

    state_type next_state(info_type) {
        return state_type();
    }

    // The second group of interfaces are used for building the table:

    static const size_t shuffle_size = shuffle_sequence_type::shuffle_size;

    bool valid_table_entry(descriptor_type desc, uint8 state_value) {
        if (state_value != 0) return false;

        // A descriptor is invalid if it contains more than three 1 bits
        // followed by a 0 bit (where we order the bits from lsb to msb).
        return bounded_unary_values_in_uint(desc, 3, 0);
    }

    uint8 num(descriptor_type desc) {
        return number_of_zeros_in_byte(desc);
    }

    uint8 prior_len(descriptor_type desc, uint8 i) {
        if (i == 0) {
            return 0;
        } else {
            uint8 p = prior_len(desc, i - 1);
            return count_trailing_ones(desc >> p) + p + 1;
        }
    }

    uint8 len(descriptor_type desc, uint8 i) {
        return count_trailing_ones(desc >> prior_len(desc, i)) + 1;
    }

    uint8 rem(descriptor_type desc) {
        return 0;
    }

    varint_g8iu_format() {
        auto shuffle_builder = cast_to_shuffle_builder(this);
        for(int i = 0; i < 256; ++i) {
            output_offsets[i] = 4 * num(descriptor_type(i));
            shuffle_builder(descriptor_type(i), 0, (int8*)(shuffles + i));
        }
    }

};
```

B.3 varint-G8CU

The `varint_g8cu_format` class describes the varint-G8CU encoding as a GroupFormat.

```
    struct varint_g8cu_format {

        typedef shuffle<32> shuffle_sequence_type;
        typedef uint8 descriptor_type;
        typedef uint16 state_type;
        typedef struct offset_and_state {
            uint8 output_offset;
            state_type next_state;
        }* info_type;

        aligned16byte_t shuffles[256*4][2];
        offset_and_state infos[256*4];

        // The first group of interfaces are used for decoding:
```

```
shuffle_sequence_type lookup_shuffle(descriptor_type desc,
                                     state_type state) {
    return shuffle_sequence_type(shuffles[state | desc]);
}

info_type lookup_info(descriptor_type desc, state_type state) {
    return infos + (state | desc);
}

ptrdiff_t input_offset(info_type) {
    return 9;  // size of data block + 1 for descriptor
}

ptrdiff_t output_offset(info_type r) {
    return ptrdiff_t(r->output_offset);
}

state_type next_state(info_type r) {
    return r->next_state;
}


// The second group of interfaces are used for table construction:

static const size_t shuffle_size = shuffle_sequence_type::shuffle_size;

bool valid_table_entry(descriptor_type desc, uint8 state_value) {
    if (state_value > 3) return false;

    // A descriptor is invalid if it contains more than three 1 bits
    // followed by a 0 bit (where we order the bits from lsb to msb),
    // including state from the prior block.
    return bounded_unary_values_in_uint(desc, 3, state_value);
}

uint8 num(descriptor_type desc) {
    return number_of_zeros_in_byte(desc);
}

uint8 prior_len(descriptor_type desc, uint8 i) {
    if (i == 0) {
        return 0;
    } else {
        uint8 p = prior_len(desc, i - 1);
        return count_trailing_ones(desc >> p) + p + 1;
    }
}
```

```
        uint8 len(descriptor_type desc, uint8 i) {
            return count_trailing_ones(desc >> prior_len(desc, i)) + 1;
        }

        uint8 rem(descriptor_type desc) {
            return leading_ones_in_byte(desc);
        }


        varint_g8cu_format() {
            auto shuffle_builder = cast_to_shuffle_builder(this);
            for(int i = 0; i < 256; ++i) {
                for (uint8 j = 0; j < 4; ++j ) {
                    descriptor_type d(i);
                    uint16 idx = ((j << 8) | d);
                    infos[idx].output_offset = 4 * num(d) + rem(d) - j;
                    infos[idx].next_state = (rem(d) << 8);
                    shuffle_builder(d, j, (int8*)(shuffles[idx]));
                }
            }
        }

    };
```

C. Bit Manipulation Functions

These bit manipulation functions are used in building some of the format-specific tables.

```
    uint8 leading_ones_in_byte(uint8 byte) {
        return uint8(count_leading_ones(0xffffff00 | uint32(byte)) - 24);
    }

    uint8 number_of_zeros_in_byte(uint8 byte) {
        return uint8(8 - popcount(byte));
    }
```

The following function returns true iff the unary values in x (consecutive 1s followed by a zero, ordering bits from lsb to msb) are bounded by b, using initial as the initial value. It is used to determine validity of descriptor and state pairs for the unary formats.

```
template<typename N>
// N  should be an unsigned integral type
bool bounded_unary_values_in_uint(N x, uint8 b, uint8 initial) {
    int adj_ones = initial;
    for(uint8 i = 0; i < 8 * sizeof(N); ++i) {
        if (ith_bit(x, i) == 0) {
            if (adj_ones > b) return false;
            adj_ones = 0;
        } else {
            ++adj_ones;
        }
    }
    return true;
}
```

D. Platform-specific Code

The following functions use intrinsics specific to a particular compiler / architecture to invoke assembly instructions.

D.1 GCC-specific code

This section contains GCC-specific implementations of functions used in the earlier algorithms. These were compiled using GCC version 4.5.1.

D.1.1 Bit Manipulation in GCC

These simple bit manipulation functions take advantage of intrinsics that GCC provides across different processor architectures.

```
int popcount(unsigned int n) {
    return __builtin_popcount(n);
}

int count_trailing_zeroes(unsigned int n) {
    return (n == 0) ? 32 : __builtin_ctz(n);
}

int count_trailing_ones(unsigned int n) {
    return count_trailing_zeroes(~n);
}

int count_leading_zeroes(unsigned int n) {
    return (n == 0) ? 32 : __builtin_clz(n);
}
```

```
int count_leading_ones(unsigned int n) {
    return count_leading_zeroes(~n);
}
```

D.1.2 Loading and PSHUFB in GCC on Intel

These functions use GCC intrinsics for SSE2 instructions that allow loading and storing 16-byte aligned quantities, where the source address (for loading) and destination (for storing) address need not be 16-byte aligned.

```
typedef char aligned16byte_t __attribute__ ((vector_size (16)));

aligned16byte_t load_unaligned128(const uint8* p) {
    return __builtin_ia32_loaddqu((const char*)(p));
}

void store_unaligned128(aligned16byte_t src, uint8* dst) {
    __builtin_ia32_storedqu((char*)(dst), src);
}
```

The following function uses GCC intrinsics for the SSSE3 PSHUFB operation.

```
void shuffle_and_store128(aligned16byte_t src,
                          const aligned16byte_t& shuffle,
                          uint8* dst) {
    store_unaligned128(__builtin_ia32_pshufb128(src, shuffle), dst);
}
```

D.2 Intel compiler-specific code

D.2.1 Bit Manipulation using Intel compiler intrinsics

These simple bit manipulation functions take advantage of intrinsics that the Intel compiler provides. They are Intel-compiler-specific implementations of the same functions as in Section D.1.1.

```
int popcount(unsigned int n) {
    return _mm_popcnt_u32(n);
}

int count_trailing_zeroes(unsigned int n) {
    return n == 0 ? 32 : _bit_scan_forward(n);
}

int count_trailing_ones(unsigned int n) {
    return count_trailing_zeroes(~n);
}

int count_leading_zeroes(unsigned int n) {
    return n == 0 ? 32 : 31 - _bit_scan_reverse(n);
}
```

```
int count_leading_ones(unsigned int n) {
    return count_leading_zeroes(~n);
}
```

D.2.2 Loading and PSHUFB using the Intel compiler

These functions use Intel compiler intrinsics for SSE2 instructions that allow loading and storing 16-byte aligned quantities, where the source address (for loading) and destination (for storing) address need not be 16-byte aligned. They are the Intel-compiler equivalents of the same functions in Section D.1.2.

```
typedef __m128i aligned16byte_t;

aligned16byte_t load_unaligned128(const uint8* p) {
    return _mm_lddqu_si128((const __m128i*)(p));
}

void store_unaligned128(aligned16byte_t src, uint8* dst) {
    _mm_storeu_si128((__m128i*)(dst), src);
}
```

The following function uses Intel compiler intrinsics for the SSSE3 PSHUFB operation.

```
void shuffle_and_store128(aligned16byte_t src,
                          const aligned16byte_t& shuffle,
                          uint8* dst) {
    store_unaligned128(_mm_shuffle_epi8(src, shuffle), dst);
}
```

References

Anh, V. N. (2004). *Impact-Based Document Retrieval*. PhD thesis, University of Melbourne.

Anh, V. N. and Moffat, A. (2005). Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166.

Apache Software Foundation (2004). Lucene 1.4.3 documentation. `http://lucene.apache.org/java/1_4_3/fileformats.html`.

Büttcher, S., Clarke, C. L. A., and Cormack, G. V. (2010a). *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, Cambridge, MA.

Büttcher, S., Clarke, C. L. A., and Cormack, G. V. (2010b). Information retrieval: Implementing and evaluating search engines, addenda for chapter 6: Index compression. `http://www.ir.uwaterloo.ca/book/addenda-06-index-compression.html`.

Croft, W. B., Metzler, D., and Strohman, T. (2010). *Search Engines: Information Retrieval in Practice*. Pearson Education, Boston.

Cutting, D. and Pedersen, J. (1990). Optimizations for dynamic inverted index maintenance. In *Proceedings of the 13th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '90, pages 405–411, New York, NY, USA. ACM.

Dean, J. (2009). Challenges in building large-scale information retrieval systems. Keynote, WSDM 2009, `http://research.google.com/people/jeff/WSDM09-keynote.pdf`.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203.

Grossman, D. A. (1995). *Integrating Structured Data and Text: A Relational Approach*. PhD thesis, George Mason University.

Heaps, H. S. (1972). Storage analysis of a compression coding for a document database. *INFOR*, 10(1):47–61.

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

Manning, C. D., Raghavan, P., and Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

MIDI Manufacturers Association (1982-2001). *MIDI 1.0 Specification*.

Power.org (2010). *Power Instruction Set Architecture*. Version 2.06 Revision B.

Schlegel, B., Gemulla, R., and Lehner, W. (2010). Fast integer compression using SIMD instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware (DaMoN 2010)*, Indianapolis, Indiana.

Stepanov, A. A. and McJones, P. (2009). *Elements of Programming*. Addison-Wesley Professional, Upper Saddle River, NJ.

Westmann, T., Kossmann, D., Helmer, S., and Moerkotte, G. (2000). The implementation and performance of compressed databases. *SIGMOD Rec.*, 29:55–67.

Wikimedia Foundation (2010). Wikipedia database download (english). `http://en.wikipedia.org/wiki/Wikipedia:Database_download`.