SIMD Compression and the Intersection of Sorted Integers

D. Lemire[1]*, L. Boytsov[2], N. Kurz[3]

[1] *LICEF Research Center, TELUQ, Montreal, QC, Canada*
[2] *Carnegie Mellon University, Pittsburgh, PA USA*
[3] *Verse Communications, Orinda, CA USA*

SUMMARY

Sorted lists of integers are commonly used in inverted indexes and database systems. They are often compressed in memory. We can use the SIMD instructions available in common processors to boost the speed of integer compression schemes. Our S4-BP128-D4 scheme uses as little as 0.7 CPU cycles per decoded 32-bit integer while still providing state-of-the-art compression.
However, if the subsequent processing of the integers is slow, the effort spent on optimizing decompression speed can be wasted. To show that it does not have to be so, we (1) vectorize and optimize the intersection of posting lists; (2) introduce the SIMD GALLOPING algorithm. We exploit the fact that one SIMD instruction can compare 4 pairs of 32-bit integers at once.
We experiment with two TREC text collections, GOV2 and ClueWeb09 (Category B), using logs from the TREC million-query track. We show that using only the SIMD instructions ubiquitous in all modern CPUs, our techniques for conjunctive queries can double the speed of a state-of-the-art approach.

KEY WORDS: performance; measurement; index compression; vector processing

1. INTRODUCTION

An inverted index maps terms to lists of document identifiers. A column index in a database might, similarly, map attribute values to row identifiers. Storing all these lists on disk can limit the performance since the latency of the fastest drives is several orders of magnitude higher than the latency of memory. Fast compression can reduce query response times [1, 2].

We assume that identifiers can be represented using 32-bit integers and that they are stored in sorted order. In this context, one can achieve good compression ratios and high decompression speed, by employing *differential coding*. We exploit the fact that the lists are sorted and instead of storing the integers themselves, we store the differences between successive integers, sometimes called the deltas.

Additional improvements arise from using the single instruction, multiple data (SIMD) instructions available on practically all server and desktop processors produced in the last decade (Streaming SIMD Extensions 2 or SSE2). A SIMD instruction performs the same operation on multiple pieces of data: this is also known as vector processing. Previous research [3] showed that we can decode compressed 32-bit integers using less than 1.5 CPU cycles per integer in a realistic inverted index scenario by using SIMD instructions. We expect that this is at least twice as fast as any non-SIMD scheme.

*Correspondence to: LICEF Research Center, TELUQ, Université du Québec, 5800 Saint-Denis, Montreal (Quebec) H2S 3L5 Canada.

Contract/grant sponsor: Natural Sciences and Engineering Research Council of Canada; contract/grant number: 261437

One downside of differential coding is that decompression requires the computation of a *prefix sum* to recover the original integers [4]: given the delta values $\delta_2, \delta_3, \ldots$ and an initial value x_1, we must compute $x_1 + \delta_2, x_1 + \delta_2 + \delta_3, x_1 + \delta_2 + \delta_3 + \delta_4 + \ldots$. When using earlier non-SIMD compression techniques, the computational cost of the prefix sum might be relatively small, but when using faster SIMD compression, the prefix sum can account for up to half of the running time. Thankfully we can accelerate the computation of the prefix sum using SIMD instructions.

Our first contribution is to revisit the computation of the prefix sum. On 128-bit SIMD vectors, we introduce four variations exhibiting various speed/compression trade-offs, from the most common one, to the 4-wise approach proposed by Lemire and Boytsov [3]. In particular, we show that at the same compression ratios, vector (SIMD) decompression is much faster than scalar (non-SIMD) decompression. Maybe more importantly, we show that *integrating* the prefix sum with the unpacking can nearly double the speed. Some of our improved schemes can decompress more than one 32-bit integer per CPU cycle. We are not aware of any similar high speed previously reported, even accounting for hardware differences.

To illustrate that SIMD instructions can significantly benefit other aspects of an inverted index (or a database system), we consider the problem of computing conjunctive queries: e.g., finding all documents that contain a given set of terms. In some search engines, such conjunctive queries are the first, and, sometimes, the most expensive, step in query processing [5, 6]. Some categories of users, e.g., patent lawyers [7], prefer to use complex Boolean queries—where conjunction is an important part of query processing.

Culpepper and Moffat showed that a competitive approach (henceforth HYB+M2) was to represent the longest lists as bitmaps while continuing to compress the short lists using differential coding [5]. To compute an intersection, the short lists are first intersected and then the corresponding bits in the bitmaps are read to finish the computation. As is commonly done, the short lists are intersected two-by-two starting from the two smallest lists: this is often called a Set-vs-Set or SvS processing. These intersections are typically computed using scalar algorithms. In fact, we are not aware of any SIMD intersection algorithm proposed for such problems on CPUs. Thus as our second contribution, we introduce new SIMD-based intersection algorithms for uncompressed integers that are up to twice as fast as competitive scalar intersection algorithms. By combining both the fast SIMD decompression and fast SIMD intersection, we can double the speed of the intersections on compressed lists.

We summarize our main contributions as follows:

1. We show that by combining SIMD unpacking and the prefix sum required by differential coding, we can improve decompression speed by up to 90 %. We end up beating by about 30 % the previous best reported decompression speeds [3].

2. We introduce a family of intersection algorithms to exploit commonly available SIMD instructions (V1, V3 and SIMD GALLOPING). They are often twice as fast as the best non-SIMD algorithms.

3. By combining our results, we nearly double the speed of a fast inverted index (HYB+M2) over realistic data and queries—on standard PC processors.

To ease reproducibility, we make all of the software and data sets publicly available, including the software to process the text collections and generate query mappings (see § 7.1 and § 7.3).

2. RELATED WORK

For an exhaustive review of fast 32-bit integer compression techniques, we refer the reader to Lemire and Boytsov [3]. Their main finding is that schemes compressing integers in large (≈ 128) blocks of integers with minimal branching are faster than other approaches, especially when using SIMD instructions. They reported using fewer than 1.5 CPU cycles per 32-bit integer on a 2011-era Intel *Sandy Bridge* processor. In comparison, Stepanov et al. [8] also proposed compression schemes optimized for SIMD instructions on CPUs, but they reported using at least 2.2 CPU cycles per 32-bit integer on a 2010 Intel *Westmere* processor.

Regarding the intersection of sorted integer lists, Ding and König [9] compared a wide range of algorithms in the context of a search engine and found that SvS with *galloping* was competitive: we describe galloping in § 6. Their own technique (RanGroup) performed better (≈10 %–30 %) but it does not operate over sorted lists but rather over a specialized data structure that divides up the data randomly into small chunks. In some instances, they found that a merge (akin to the merge step in the merge sort algorithm) was faster. They also achieved good results with Lookup [10]: a technique that relies on an auxiliary data structure for skipping values [11]. Ding and König found that alternatives such as Baeza-Yates' algorithm [12] or adaptive algorithms [13] were slower.

Barbay et al. [14] also carried out an extensive experimental evaluation. On synthetic data using a uniform distribution, they found that Baeza-Yates' algorithm [12] was faster than SvS with galloping (by about 30 %). However, on real data (e.g., TREC GOV2), SvS with galloping was superior to most alternatives by a wide margin (e.g., 2× faster).

Culpepper and Moffat similarly found that SvS with galloping was the fastest [5] though their own `max` algorithm was fast as well. They found that in some specific instances (for queries containing 9 or more terms) a technique similar to galloping (interpolative search) was slightly better (by less than 10 %).

Kane and Tompa improved Culpepper and Moffat's HYB+M2 by adding auxiliary data structures to skip over large blocks of compressed values (256 integers) during the computation of the intersection [15]. Their good results are in contrast with Culpepper and Moffat's finding that skipping is counterproductive when using bitmaps [5].

Our work is focused on commodity desktop processors. Compression and intersection of integer lists using a graphics processing unit (GPU) has also received attention. Ding et al. [16] improved the intersection speed using a *parallel merge find*: essentially, they divide up one list into small blocks and intersect these blocks in parallel with the other array. On conjunctive queries, Ding et al. [16] found their GPU implementation to be only marginally superior to a CPU implementation (≈15 % faster) despite the data was already loaded in GPU's global memory. They do, however, get impressive speed gains (7×) on disjunctive queries.

Ao et al. [17] proposed a parallelized compression technique (Parallel PFor) and replaced conventional differential coding with an approach based on linear regression. In our work, we rely critically on differential coding, but alternative models merit consideration [18, 19, 20].

3. RELEVANT SIMD INSTRUCTIONS

Intel PC processors support vector instructions, but languages such as C or C++ do not directly include vectorization as part of their standard syntax. However, it is still possible to conveniently call these instructions by using intrinsics or online assembly code. Intrinsics are special functions (sometimes called built-in functions) provided as a compiler extension of C/C++ syntax.

Table I presents the various SIMD instructions we require. In addition to instruction mnemonic names, we also provide names of respective C/C++ intrinsics. In this table, there is a single intrinsic function per mnemonic except for the move instruction `movdqu`, which has separate intrinsics to denote store and load operations.

Though most of the instructions are fast, some are significantly more expensive. Intel often expresses the computational cost of an instruction in terms of its latency and reciprocal throughput. The latency is the minimum number of cycles required to execute the instruction. The reciprocal throughput is one over the maximum number of instructions of the same kind that can be executed per cycle. For example, a reciprocal throughput of 0.5 indicates that up to two instructions of the same type can be executed in a cycle. We give the latency and reciprocal throughput for Intel processors with a *Sandy Bridge* microarchitecture [21]. These numbers are often constant across Intel microarchitectures.

All of these instructions use 128-bit registers (called *XMM registers*). Many of them are straightforward. For example, `por` and `pand` compute the bitwise OR and AND between two registers.

We use the `movdqu` instruction to load or store a register. Loading and storing registers has a relatively high latency (3 cycles). While we can load two registers per cycle, we can only store one of them to memory.

The bit shifting instructions come in two flavors. When we need to shift entire registers by a number of bits divisible by eight (a byte), we use the `psrldq` and `pslldq` instructions. They have a high throughput (2 instructions per cycle) on *Sandy Bridge* Intel processors [22].

We can also consider the 128-bit registers as a vector of four 32-bit integers. We can then shift right four 32-bit integers by a fixed number of bits using the `psrld` instruction. It has reduced throughput compared to the byte-shifting instructions (1 instruction per cycle).

In the same spirit, we can add four 32-bit integers with four other 32-bit integers at once using the `paddd` instruction. It is another fast operation.

Sometimes it is necessary to copy the 32-bit integers from one XMM register to another, while possibly moving or duplicating values. The `pshufd` instruction can serve this purpose. It takes as a parameter an input register v as well as a control mask m. The control mask is made of four 2-bit integers each representing an integer in $\{0, 1, 2, 3\}$. We output $(v_{m_0}, v_{m_1}, v_{m_2}, v_{m_3})$. Thus, for example, the `pshufd` instruction can copy one particular value to all positions (using a mask made of 4 identical values). It is a fast instruction with a throughput of two instructions per cycle.

We can compare the four 32-bit integers of one register v with the four 32-bit integers of another register v' using the `pcmpeqd` instruction. It generates four 32-bit integers with value 0xFFFFFFFF or 0 depending on whether the corresponding pairs of integers are equal (e.g., if $v_0 = v'_0$, then the first component returned by `pcmpeqd` has value 0xFFFFFFFF).

The `pcmpeqd` instruction can be used in tandem with the `movmskps` instruction, which generates a 4-bit mask by extracting the four most significant bits from the four 32-bit integers produced by the comparison instruction `pcmpeqd`. The `movmskps` instruction is slightly expensive, with a throughput of 1 instruction per cycle and a latency of 2 cycles. In some cases, we only need to verify if four 32-bit integers in one register are different from respective 32-bit integers in another register, i.e., we do not need to extract a mask. To check this, we use either the SSE2 instruction `pmovmskb` or the SSE4 instruction `ptest`. The SSE4 instruction has smaller latency, however, replacing `ptest` with `pmovmskb` did not substantially affect runtime.

In some particular algorithms (see § 6.1), we also use two recently introduced string-comparison instructions (part of the SSE4 instruction set): `pcmpestrm` and `pcmpistrm`. They operate on 8-bit or 16-bit strings loaded in XMM registers. They take a control mask to specify their behavior. We use them for 16-bit strings. For our purposes, we ask for a 8-bit mask indicating whether any of the eight 16-bit elements of the first register are equal to any of the eight 16-bit elements of the second register. This mask is stored in a 128-bit MMX register and is extracted using the `pextrd` instruction.

The two string-comparison instructions differ in how they deduce the string length. The slower `pcmpestrm` instruction requires us to specify the string lengths (8 elements in our case). The faster `pcmpistrm` instruction assumes that strings are null terminated. When no null value is found, the `pcmpistrm` instruction processes 8 elements per register. There is no 16-bit counterpart to the `pshufd` instruction, but there is an 8-bit version (`pshufb`) with the same latency and throughput.

4. INTEGER COMPRESSION

We consider the case where we have lists of integers stored using 32 bits, but where the magnitude of most integers requires fewer than 32-bits to express. We want to compress them while spending as few CPU cycles per integer as possible.

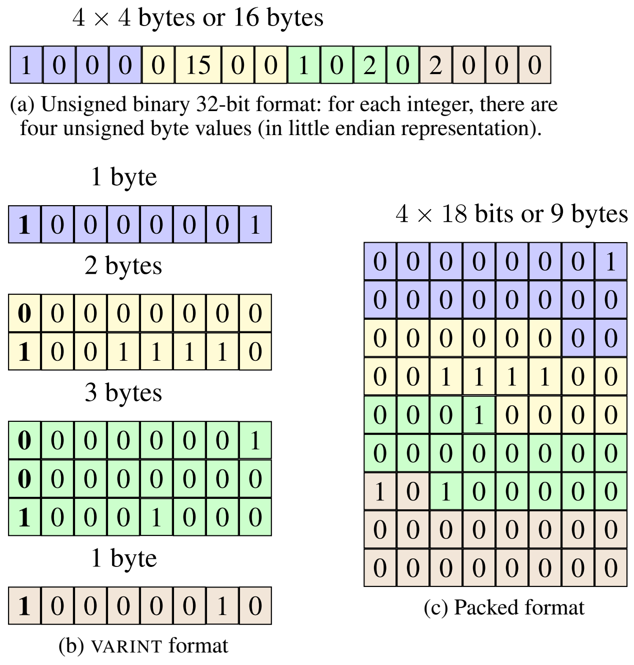
Table I. SIMD instructions on *Sandy Bridge* Intel processors with latencies and reciprocal throughput in CPU cycles . All instructions are part of SSE2, except `pcmpestrm`, `pcmpistrm`, `pextrd`, `popcnt`, `ptest` that are SSE4 instructions, and `pshufb` that is an SSSE3 instruction.

instruction	C/C++ intrinsic	description	latency	rec. thr.
`por`	`_mm_or_si128`	bitwise OR	1	0.33
`pand`	`_mm_and_si128`	bitwise AND	1	0.33
`movdqu`	`_mm_storeu_si128`	store a 128-bit register	3	1
`movdqu`	`_mm_loadu_si128`	load to 128-bit register	3	0.5
`psrldq`	`_mm_srli_si128`	shift right by a number of bytes	1	0.5
`pslldq`	`_mm_slli_si128`	shift left by a number of bytes	1	0.5
`psrld`	`_mm_srl_epi32`	shift right four 32-bit integers	1	1
`pslld`	`_mm_sll_epi32`	shift left four 32-bit integers	1	1
`paddd`	`_mm_add_epi32`	add four 32-bit integers	1	0.5
`pshufd`	`_mm_shuffle_epi32`	shuffle four 32-bit integers	1	0.5
`pcmpeqd`	`_mm_cmpeq_epi32`	compare four 32-bit integers for equality	1	0.5
`movmskps`	`_mm_movemask_ps`	mask from most significant bits of 32-bit elements	2	1
`pmovmskb`	`_mm_movemask_epi8`	mask from most significant bits of 16-bit elements	2	1
`pcmpestrm`	`_mm_cmpestrm`	compare two strings of specific lengths	12	4
`pcmpistrm`	`_mm_cmpistrm`	compare two null-terminated strings	11	3
`pshufb`	`_mm_shuffle_epi8`	shuffle 16 bytes	1	0.5
`pextrd`	`_mm_extract_epi32`	extract a specified 32 bit integer	2	1
`popcnt`	`_mm_popcnt_u32`	number of 1s in a 32-bit integer	3	1
`ptest`	`_mm_testz_si128`	performs a bitwise and of two 128-bit integers; returns one, if the result is all zeros, and zero otherwise.	1	1

There has been much work on the design and analysis of integer compression schemes. Out of an earlier survey [3], we choose 4 previously described fast compression schemes: VARINT, S4-BP128, FASTPFOR and S4-FASTPFOR. Both S4-BP128 and S4-FASTPFOR performed best in an exhaustive experimental comparison [3]. We review them briefly for completeness.

4.1. VARINT

Many authors such as Culpepper and Moffat [5] use variable byte codes (henceforth VARINT) also known as escaping [10] for compressing integers. It was first described by Thiel and Heaps [23]. For example, we might code integers in $[0, 2^7)$ using a single byte, integers in $[2^7, 2^{14})$ using two bytes and so on. As an example, consider the integers 1, 3840, 131073, and 2, and Fig. 1. In Fig. 1a, we give the usual 32-bit integer layout for these integers, it uses 16 bytes. In binary format, these numbers can be written 1, 111100000000, 100000000000000001 and 10 (writing the most significant bits first). The first integer can be written using a single byte since it is in $[0, 2^7)$. To indicate that the byte corresponds to a complete integer, we set the most significant bit to 1: 10000001. The integer 3840 is in $[2^7, 2^{14})$, so we represent it using two bytes. The first byte corresponds to the first 7 bits and has 0 as its most significant bit to indicate that it does not represent a full integer (000000001). The second byte includes the remaining bits and has 1 as its most significant bit (10011110). We proceed similarly for the integers 131073 and 2 (see Fig. 1b). Overall, VARINT using 7 bytes instead of the original 16 bytes. VARINT does not always compress well: it



(a) Unsigned binary 32-bit format: for each integer, there are four unsigned byte values (in little endian representation).

(b) VARINT format

(c) Packed format

Figure 1. The sequence of numbers 1, 3840, 131073 and 2 using 3 different data formats. We use a left to right, top to bottom representation, putting most significant bits first in each byte.

always uses at least one byte per integer. However, if most integers can be represented with a single byte, then it offers competitive decompression speed [3]. Stepanov et al. [8] reviewed VARINT-like alternatives that use SIMD instructions. Lemire and Boytsov [3] found that the fastest such alternative (varint-G8IU) was slower and did not compress as well as other SIMD-based schemes.

4.2. *Bit packing and unpacking*

Consider once more the integers 1, 3840, 131073, and 2, and Fig. 1. The largest of them is 131073 and it can be represented using 18 bits in binary format (as 100000000000000001). Thus we could decide to write the four integers using exactly 18 bits per integer. The result in memory might look like Fig. 1c and span 9 bytes. We call this process *bit packing*, and the reverse process *bit unpacking* (see Table II for a summary of our terminology).

The consecutive layout of Fig. 1 is adequate for scalar processing: unpacking an integer can be done efficiently using shifts and bitwise logical operations. However, when using SIMD processing, we want to pack and unpack several (e.g., 4) integers at once. For this reason, we choose to interleave the packed integers as in Fig. 2. In this example, the integers x_1, x_2, x_3, x_4 are packed to the first 18 bits of 4 consecutive 32-bit words. The first 14 bits of the next integers x_5, x_6, x_7, x_8 are packed to the next 14 bits of these 32-bit integers, and the remaining $18 - 14 = 4$ bits are packed to the first bits of the next four consecutive 32-bit words. See Lemire and Boytsov [3] for a more detailed discussion and a comparison with an alternative layout.

Only 4 basic operations are required for bit unpacking: bitwise or, bitwise and, logical shift right, and logical shift left. The corresponding 128-bit SSE2 instructions operating on packed 32-bit integers are `por`, `pand`, `psrld`, and `pslld` (in Intel and AMD processors). For a given bit width b, no branching is required for bit packing or bit unpacking. Thus, we can create one bit unpacking function for each bit width b and select the desired one using an array of function pointers or a `switch/case` statement. In the scalar case, it is most convenient to pack and unpack integers in units of 32 integers. Given a bit width b, the unpacking procedure outputs b 32-bit integers. In the vectorized case, we pack and unpack integers in blocks of 128 integers, so that

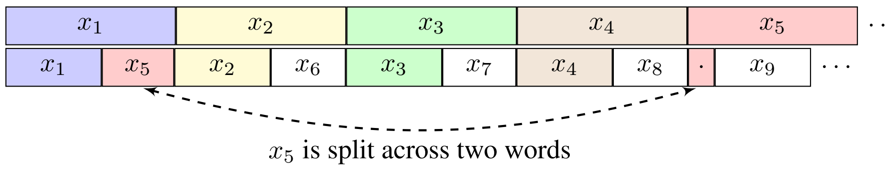


Figure 2. Packing 32-bit integers $x_1, x_2, \ldots$ (top) to 18 bits per integer (bottom) using the interleaved packed format for SIMD processing

the unpacking procedure–corresponding to bit width b–generates b 128-bit vectors. A generic bit unpacking procedure for a block of 128 integers is given by Algorithm 1 and discussed again in § 5.

4.3. S4-BP128

Bit packing suggests a simple scheme: regroup the integers into blocks (e.g. 128 integers) and pack them as concisely as possible, while recording the bit width (e.g., 18 bits per integer) using an extra byte. We call this approach BINARY PACKING. Binary packing is closely related to Frame-Of-Reference (FOR) [24] and it has been called PackedBinary [25].

Binary packing can be fast. Indeed, Catena et al. [1] found that it offered the best speed in a search engine setting. Our fastest family of compression schemes is an instance of binary packing: S4-BP128. The “S4” stands for 4-integer SIMD, “BP” stands for “Binary Packing”, and “128” indicates the number of integers encoded in each block.

In the S4-BP128 format, we decompose arrays into meta-blocks of 2048 integers, each containing 16 blocks of 128 integers. Before bit packing the 16 blocks, we write 16 bit widths (b) using one byte each. For each block, the bit width is the smallest value b such that all corresponding integers are smaller than 2^b. The value b can range from 0 to 32. In practice, lists of integers are rarely divisible by 2048. We handle remaining blocks of 128 integers separately. For each such block, we write a byte containing a bit width b, followed by the corresponding 128 integers in bit packed form. Finally, we compress the remaining integers (less than 128 integers) using VARINT.

4.4. FASTPFOR, SIMD-FASTPFOR, *and* S4-FASTPFOR

The downside of the S4-BP128 approach is that the largest integers in a block of 128 integers determine the compression ratio of all these integers. We could use smaller blocks (e.g., 32) to improve compression. However, a better approach for performance might be *patching* [26] wherein the block is first decompressed using a smaller bit width, and then a limited number of entries requiring greater bit widths are overwritten (“patched”) using additional information. That is, instead of picking the bit width b such that all integers are smaller than 2^b, we pick a different bit width b' that might be smaller than b. That is, we only bit pack the least significant b'-bits from each integer. We must still encode the missing information for all integers larger than or equal to $2^{b'}$: we call each such integer an exception.

In our version of patched coding (FASTPFOR, for “fast patched frame-of-reference”), we proceed as in S4-BP128, that is we bit pack blocks of 128 integers. To complement this data, we use a *metadata* byte array. For each block, we store both bit widths b and b', as well as the number of exceptions. Each location where an exception should be applied is also stored using one byte in the *metadata* byte array. It remains to store the most significant $b - b'$ bits of the integers larger than or equal to $2^{b'}$. We collect these exceptions over many blocks (up to 512 blocks) and then bit pack them together, into up to 32 bit packed arrays (one for each possible value of $b - b'$ excluding zero). For speed, these arrays are padded up to a multiple of 32 integers. The final data format is an aggregation of the bit packed blocks (using b' bits per integer), the *metadata* byte array and the bit packed arrays corresponding to the exceptions.

As an illustration, consider Fig. 3. We consider a block of 128 integers beginning with 1,2, 1, 134217729, 0. We assume that all values in the block except for 134217729 do not exceed three and, thus, can be encooded using two bits each. In this case, we have that $b = 27$ and binary packing

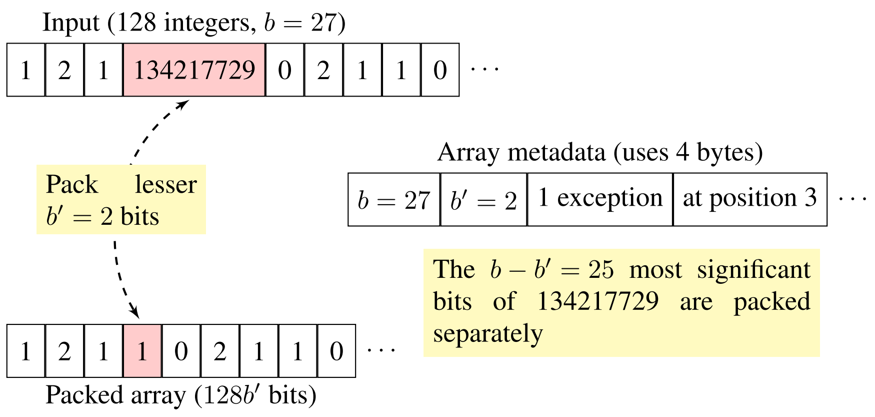

Input (128 integers, $b = 27$)
1 2 1 134217729 0 2 1 1 0 ···
Pack lesser $b' = 2$ bits
Array metadata (uses 4 bytes)
$b = 27$ | $b' = 2$ | 1 exception | at position 3 ···
The $b - b' = 25$ most significant bits of 134217729 are packed separately
1 2 1 1 0 2 1 1 0 ···
Packed array ($128b'$ bits)

Figure 3. Example of compression using FASTPFOR (and S4-FASTPFOR). The data corresponding to a block of 128 integers is located in a packed array (using b' bit per integer, in a metadata byte array and, finally, in a corresponding packed array of exceptions.

would thus require 27 bits per integer—even if most of the integers could fit in two bits. With FASTPFOR, we might set $b' = 2$ and pack only the least significant two bits of each integer. To allow decoding, we have to record that there is one exception at location 3 as well as the values of b and b'. Each of these numbers can be stored using one byte. We are left to code the $b - b' = 25$ most significant bits of 134217729. We can store these bits in an array to be packed later. Once we have processed many blocks of 128 integers, there might be several similar exceptions that need the storage of their 25 most significant bits: they are stored and packed together. During decompression, these most significant bits can be unpacked as needed from one of 32 different arrays (one for each possible value of $b - b'$ excluding zero).

Decompression using FASTPFOR is similar to decompression using S4-BP128, except that, after bit unpacking the least significant b' bits for the integers of a block, we must proceed with the *patching*. For this purpose, the packed arrays of exception values are unpacked as needed and we loop over the exceptions.

The integer b is determined by the data—as the number of bits required to store the largest integer in the block. However, we are free to set b' to any integer in $[0, b]$. For each block, we pick b' to minimize $128 \times b' + c(b')(b - b' + 8)$ where $c(b')$ is the number exceptions generated for a given value b' (e.g., $c(b) = 0$). This heuristic for picking b' can be computed quickly if we first tabulate how many integers in the block are less than 2^x for $x = 0, \ldots, b$. For this purpose, we use the assembly instruction `bsr` (Bit Scan Reverse) to calculate the $\log_2$ of each integer.

We can compare FASTPFOR with other patched schemes based on Lemire and Boytsov's results [3]: the original scheme from Zukowski et al. [26] is faster than FASTPFOR but has worse compression ratios (even worse than the bit packing method S4-BP128 that does not use patching), OptPFD [27] by Yan el al. sometimes compresses better than FASTPFOR, but can be up to two times slower, and their NewPFD [27] is typically slower and offers worse compression ratios than FASTPFOR. A format inspired by FASTPFOR (where bits from exceptions are bit packed) is a part of the search engine Apache Lucene as of version 4.5.[†] In contrast, Catena et al. [1] found that FASTPFOR provided no response time benefit compared to NewPFD and OptPFD when compressing document identifiers. Possibly, the discrepancy can be explained by the fact that they divided their arrays into small chunks of 1024 integers prior to compression. Indeed, FASTPFOR is most effective when it is allowed to aggregate the exceptions over many blocks of 128 integers.

Because FASTPFOR relies essentially on bit packing for compression, it is easy to vectorize it using the same SIMD-based bit packing and unpacking functions used by S4-BP128. We call the

[†]http://lucene.apache.org/core/4_9_0/core/org/apache/lucene/util/PForDeltaDocIdSet.html

Table II. Some of our terminology

Differential coding	Strategy to transform lists of sorted integers into lists of small integers by computing differences between nearby, but not necessarily adjacent, integers. We consider four types of differential coding (D1, D2, DM, D4). To recover the initial array, it is necessary to compute a prefix sum or its analog. Optionally, we can integrate differential coding in the packing and unpacking. In such a case, unpacking includes the computation of the prefix sum.
pack (unpack)	To encode (resp. decode) 32-bit integers to (resp. from) data blocks where all integers use a fixed number of bits (bit width). Packing and unpacking routines are used as part of integer compression algorithms such as binary packing.
binary packing	Compression scheme that packs blocks of integers (e.g., 128 integers) using as few bits as possible (e.g., the S4-BP128 family)
patched coding	Compression strategy that uses binary packing to encode most of the data, while also encoding exceptions separately (e.g., the FASTPFOR scheme) to improve the compression ratio.

resulting scheme S4-FASTPFOR. In the original scheme by Lemire and Boytsov [3] (called SIMD-FASTPFOR), the bit-packed exceptions were padded up to multiples of 128 integers instead of multiples of 32 integers as in FASTPFOR. While this insured that all memory pointers were aligned on 16 bytes boundaries, it also adversely affected compression. Our S4-FASTPFOR has essentially the same data format as FASTPFOR and, thus, the same compression ratios. These changes require that we use (1) scalar bit packing/unpacking for up to 32 integers per packed array and (2) unaligned load and store SIMD instructions. Neither of these differences significantly impacts performance, and the compression ratios improve by $\approx 5\,\%$.

5. DIFFERENTIAL CODING

Differential coding takes a sorted integer list $x_1, x_2, \ldots$ and replaces it with successive differences (or deltas) $\delta_2, \delta_3, \ldots = x_2 - x_1, x_3 - x_2, \ldots$. If we keep the first value intact (x_1), this transformation is invertible. We can then apply various integer compression schemes on the deltas. Since the deltas are smaller than the original integers in the list, we get better compression ratios.

If the sorted lists contain no repetitions, we can further subtract one from all deltas as we know that they are greater than zero. However, this only offers a significant benefit in compression if we expect many of the deltas to be close to zero. And, in such cases, other schemes such as bit vectors might be more appropriate. Thus, we do not consider this option any further.

Computing deltas during compression is an inexpensive operation that can be easily accelerated with superscalar execution or even SIMD instructions. However, recovering the original list from the deltas when decompressing can be more time consuming because of the inherent data-dependencies: the value of each recovered integer can only be calculated after the value of the integer immediately preceding it is known. Indeed, it involves the computation of a prefix sum: $x_i = x_{i-1} + \delta_i$. A naive implementation could end up using one or more CPU cycles per integer just to calculate the prefix sum. For a moderately fast scheme such as VARINT, this is not a concern, but for faster schemes, computation of the prefix sum can become a performance bottleneck. To alleviate this problem, we can sacrifice some compressibility, by computing the deltas on a four-by-four basis (henceforth D4, because it compares index differences of 4): $\delta_i = x_i - x_{i-4}$ [3]. Although fast, this approach also generates larger deltas.

As further refinements, we consider four different forms of differential coding that offer different compression/speed trade-offs. Each form has a corresponding inverse—for simplicity, we use the term *prefix sum* to refer to all these inverse functions. As before, we assume 128-bit vectors and process 32-bit integers.

- The fastest is D4 which computes the deltas four-by-four: e.g., $(\delta_5, \delta_6, \delta_7, \delta_8) = (x_5, x_6, x_7, x_8) - (x_1, x_2, x_3, x_4)$. We expect the deltas to be $4\times$ larger, which degrades the compression by approximately two bits per integer. However, a single 1-cycle-latency SIMD instruction (`paddd` in SSE2) can correctly calculate the prefix sum of four consecutive integers.
- The second fastest is DM. It is similar to D4 except that instead of subtracting the previously decoded vector of integers, we subtract only the largest of these integers: $\delta_{4i+j} = x_{4i+j} - x_{4i-1}$. E.g.,

  $$(\delta_5, \delta_6, \delta_7, \delta_8) = (x_5, x_6, x_7, x_8) - (x_4, x_4, x_4, x_4).$$

  We expect the deltas to be $2.5\times$ larger on average. Compared to the computation of the prefix sum with D4, DM requires one extra instruction (`pshufd` in SSE2) to copy the last component to all components: $(x_1, \ldots, x_4) \to (x_4, \ldots, x_4)$. On Intel processors, the `pshufd` instruction is fast.
- The third fastest is D2: $\delta_i = x_i - x_{i-2}$. E.g.,

  $$(\delta_5, \delta_6, \delta_7, \delta_8) = (x_5, x_6, x_7, x_8) - (x_3, x_4, x_5, x_6).$$

  The deltas should be only $2\times$ larger on average. The prefix sum for D2 can be implemented using 4 SIMD instructions.

  1. Shift the delta vector by 2 integers (in SSE2 using `psrldq`): e.g., $(\delta_5, \delta_6, \delta_7, \delta_8) \to (0, 0, \delta_5, \delta_6)$.
  2. Add the original delta vector with the shifted version: e.g., $(\delta_5, \delta_6, \delta_5 + \delta_7, \delta_6 + \delta_8)$.
  3. Select from the previous vector the last two integers and copy them twice (in SSE2 using `pshufd`), e.g., $(x_1, x_2, x_3, x_4) \to (x_3, x_4, x_3, x_4)$.
  4. Add the results of the last two operations.
- The slowest approach is D1 which is just the regular differential coding ($\delta_i = x_i - x_{i-1}$). It generates the smallest deltas. We compute it with a well-known approach using 6 SIMD instructions.

  1. The first two steps are as with D2 to generate $(\delta_5, \delta_6, \delta_5 + \delta_7, \delta_6 + \delta_8)$. Then we take this result, shift it by one integer and add it to itself. Thus, we get:

     $$(\delta_5, \delta_5 + \delta_6, \delta_6 + \delta_5 + \delta_7, \delta_5 + \delta_6 + \delta_7 + \delta_8).$$
  2. We copy the last integer of the previous vector to all components of a new vector. Thus, we generate (x_4, x_4, x_4, x_4).
  3. We add the last two results to get the final answer.

We summarize the different techniques (D1, D2, DM, D4) in Table III. We stress that the number of instructions is not a measure of running-time performance, if only because of superscalar execution. Our analysis is for 4-integer SIMD instructions: for wider SIMD instructions, the number of instructions per integer is smaller.

Table III. Comparison between the 4 vectorized differential coding techniques with 4-integer SIMD instructions

	size of deltas	instructions/int
D1	$1.0\times$	1.5
D2	$2.0\times$	1
DM	$2.5\times$	0.5
D4	$4.0\times$	0.25

Combining a compression scheme like VARINT and differential coding is not a problem. Since we decode integers one at a time, we can easily integrate the computation of the prefix sum into the decompression function: as soon as a new integer is decoded, it is added to the previous integer.

For S4-BP128 and S4-FASTPFOR, we could integrate differential coding at the block level. That is, we could unpack 128 deltas and then calculate the prefix sum to convert these deltas back to the original integers. Though the decompression requires two passes over the same small block of integers, it is unlikely to cause many expensive cache misses.

Maybe surprisingly, we can do substantially better, at least for schemes such as S4-BP128. Instead of using two passes, we can use a single pass where we do both the bit unpacking and the computation of the prefix sum. In some cases, the one-pass approach is almost twice as fast as the two-pass approach.

Indeed, when unpacking a block of 128 integers, we store 32 SSE registers in memory. Such store operations have a limited throughput of one per cycle on recent Intel processors [21]. However, if the prefix sum computation is a separate step, we need to reload the recently unpacked data, compute the prefix sum and then store the data again. In effect, we end up having to store 64 SSE registers per block of 128 integers. Hence, we need at least 0.5 cycles to unpack an integer if the prefix sum computation is separate. These store operations can become a bottleneck. A two-step unpacking has a theoretical speed limit of 7.2 billion 32-bit integers per second on a 3.60 GHz processor. Yet we can nearly reach 9 billion 32-bit integers per second with a one-step unpacking (see § 7.4).

Henceforth we use the term unpacking to also refer to the process where we both unpack and compute the prefix sum. Algorithm 1 illustrates the bit unpacking routine for a block of 128 integers. It takes as a parameter a SIMD prefix-sum function P used at lines 10 and 16: for D4, we have $P(t, v) = t + v$ (an element-wise addition of two vectors). Omitting lines 10 and 16 disables differential coding. In practice, we generate one such function for each bit width b and for each prefix-sum function P. The prefix sum always starts from an initial vector (v). We set $v = (0, 0, 0, 0)$ initially and then, after decompressing each block of 128 integers, v is set to the last 4 integers decoded.

Beside the integration of differential coding with the bit unpacking, we have also improved over Lemire and Boytsov's bit unpacking [3, Fig. 7] in another way: whereas each of their procedures may require several masks, our implementation uses a single mask per procedure (see line 4 in Algorithm 1). Given that we only have 16 SIMD registers on our Intel processors, attempting to keep several of them occupied with constant masks can be wasteful.

Unfortunately, for S4-FASTPFOR, it is not clear how to integrate bit unpacking and computation of the prefix sum. Indeed, S4-FASTPFOR requires three separate operations in sequence: bit unpacking, patching and computing the prefix sum. Patching must happen after bit unpacking, but before the prefix sum. This makes tight integration between patching and the prefix sum difficult: we tried various approaches but they did not result in performance improvement.

6. FAST INTERSECTIONS

Consider lists of uncompressed integers. To compute the intersection between several sorted lists quickly, a competitive approach is Set-vs-Set (SvS): we sort the lists in the order of non-decreasing cardinality and intersect them two-by-two, starting with the smallest. A textbook intersection

Algorithm 1 Unpacking procedure using 128-bit vectors with integrated differential coding. We write $\gg$ for the bitwise *zero-fill* right shift, $\ll$ for the bitwise left shift, $\&$ for the bitwise AND, and $|$ for the bitwise OR. The binary function P depends on the type of differential coding (e.g., to disable differential coding set $P(t, v) = t$).

```
 1: input: a bit width $b$, a list of 32-bit integers $y_1, y_2, \ldots, y_b$, prefix-sum seed vector $v$
 2: output: list of 128 32-bit integers in $[0, 2^b)$
 3: $w \leftarrow$ empty list
 4: $M \leftarrow (2^b - 1, 2^b - 1, 2^b - 1, 2^b - 1)$ {Reusable mask}
 5: $i \leftarrow 0$
 6: for $k = 0, 1, \ldots, b - 1$ do
 7:     while $i + b \leq 32$ do
 8:         $t \leftarrow (y_{1+4k} \gg i, y_{2+4k} \gg i, y_{3+4k} \gg i, y_{4+4k} \gg i)$
 9:         $t \leftarrow t \;\&\; M$ {Bitwise AND with mask}
10:         $t \leftarrow P(t, v)$ and $v \leftarrow t$ {Prefix sum}
11:         append integers $t_1, t_2, t_3, t_4$ to list $w$
12:         $i \leftarrow i + b$
13:     if $i < 32$ then
14:         $z \leftarrow (y_{5+4k} \ll 32 - i, y_{6+4k} \ll 32 - i,$
                $y_{7+4k} \ll 32 - i, y_{8+4k} \ll 32 - i) \;\&\; M$
15:         $t \leftarrow (y_{1+4k} \gg i, y_{2+4k} \gg i,$
                $y_{3+4k} \gg i, y_{4+4k} \gg i) \mid z$
16:         $t \leftarrow P(t, v)$ and $v \leftarrow t$ {Prefix sum}
17:         append integers $t_1, t_2, t_3, t_4$ to list $w$
18:         $i \leftarrow i + b - 32$
19:     else
20:         $i \leftarrow 0$
21: return $w$
```

algorithm between two lists (akin to the merge sort algorithm) runs in time $O(m + n)$ where the lists have length m and n (henceforth we call it SCALAR). Though it is competitive when m and n are similar, there are better alternatives when $n \gg m$. Such alternative algorithms assume that we intersect a small list r with a large list f. They iterate over the small list: for each element r_i, they seek a match f_j in the second list using some search procedure. Whether there is a match or not, they advance in the second list up to the first point where the value is at least as large as the one in the small list ($f_j \geq r_i$). The search procedure assumes that the lists are sorted so it can skip values. A popular algorithm uses galloping search [28] (also called exponential search): we pick the next available integer r_i from the small list and seek an integer at least as large in the other list, looking first at the next available value, then looking twice as far, and so on. We keep doubling the distance until we find the first integer $f_j \geq r_i$ that is not smaller than r_i. Then, we use binary search to advance in the second list to the first value larger than or equal to r_i. The binary search range is from the current position to the position of value f_j. Computing galloping intersections requires only $O(m \log n)$ time which is better than $O(m + n)$ when $n \gg m$.

6.1. Existing SIMD intersection algorithms

We could find only two algorithms for the computation of intersections over sorted uncompressed lists of integers using SIMD instructions.

1. Schlegel et al. [29] use a specialized data structure, where integers are partitioned into sub-arrays of integers having the same 16 most significant bits. To represent sub-array values, one needs 16 bits for each sub-array element plus 16 bits to represent the most significant bytes shared among all the integers.

To compute the intersection between two lists, Schlegel et al. iterate over all pairs of 16-bit sub-arrays (from the first and the second list, respectively) with identical 16 most significant bits. In each iteration, they compute the intersection over two 16-bit arrays. To this end, each sub-array is logically divided into blocks each containing eight 16-bit integers. For simplicity, we assume that the number of integers is a multiple of eight. In practice, lists may contain a number of integers that is not divisible by eight. However, one can terminate the computation of the intersection with the few remaining integers and process these integers using a more traditional intersection algorithm.

The intersection algorithm can be seen as a generalization of the textbook intersection algorithm (akin to the merge sort algorithm). There are two block pointers (one for each sub-array) originally pointing to the first blocks. Using SIMD instructions, it is possible to carry out an all-against-all comparison of 16-bit integers between these two blocks and extract matching integers. Schlegel et al. exploit the fact that such a comparison can quickly be done using the SSE 4.1 string-comparison instruction `pcmpestrm`. (See § 3.) Recall that this instruction takes strings of up to eight 16-bit integers and returns an 8-bit mask indicating which 16-bit integers are present in both strings. The number of 1s in the resulting mask (computed quickly with the `popcnt` instruction) indicates the number of matching integers. Schlegel et al. apply this instruction to two blocks of eight integers (one from the first list and another from the second list). To obtain matching integers, they use the shuffle SSSE3 instruction `pshufb`. This instruction extracts and juxtaposes matching integers so that they follow each other without gaps. This requires a lookup table to convert 8-bit masks produced by `pcmpestrm` (and extracted using `pextrd`) to the shuffle mask for `pshufb`. The lookup table contains 256 128-bit shuffle masks. The matching integers are written out and the output pointer is advanced by the number of matching integers.

Afterwards, the highest values of the two blocks are compared. If the highest value in the first block is smaller than the highest value of the second block, the first block pointer is advanced. If the second block contains a smaller highest value, its pointer is advanced in a similar fashion. If the highest values are equal, the block pointers of both lists are advanced. When the blocks in one of the lists are exhausted, the intersection has been computed.

We slightly improved Schlegel et al.'s implementation by replacing the `pcmpestrm` instruction with the similar `pcmpistrm` instruction. The latter is faster. Its only downside is that it assumes that the two strings of 16-bit characters are either null terminated or have a length of 8 characters. However, because the integers are sorted (even in 16-bit packed blocks), we can easily check for the presence of a zero 16-bit integer as a special case.

Schlegel et al. only validated their results on the intersection of arrays that have identical lengths. In contrast, we work on arrays of 32-bit integers having differing lengths.

2. Katsov [30] proposed an approach similar to Schlegel et al. except that it works on arrays of 32-bit integers. We process the input arrays by considering blocks of four 32-bit integers. Given two blocks (a, b, c, d) and (a', b', c', d'), we execute four comparisons:

   - $(a, b, c, d) = (a', b', c', d')$,
   - $(a, b, c, d) = (b', c', d', a')$,
   - $(a, b, c, d) = (c', d', a', b')$,
   - $(a, b, c, d) = (d', a', b', c')$.

   For each comparison, the `pcmpeqd` instruction generates four integers with value 0xFFFFFFFF or 0 depending on whether the corresponding pairs of integers are equal. By computing the bitwise OR of the 4 resulting masks, we know whether each value in (a, b, c, d) matches one value in (a', b', c', d'). We can extract the corresponding 4-bit mask with the `movmskps` instruction. The rest is similar to Schlegel et al.'s algorithm.

There has been extensive work to accelerate sort or search algorithms by using SIMD instructions [31, 32], among others. Nevertheless, we are not aware of other algorithms practically relevant to the intersection of sorted lists of integers using SIMD instructions.

6.2. *Our intersection algorithms*

Both the SvS and textbook algorithms involve checking whether an element from one list is in the other list. When one list is substantially shorter than the other, we typically compare a single integer from the shorter list against several adjacent integers from the longer list. Such comparisons can be efficiently done using SIMD operations (see sample code in Appendix A) and this is the underlying idea of our intersection algorithms. Unlike the algorithms by Schlegel et al. and Katsov we do not need to process a mask to determine which integers intersect since we check only one integer at a time from the short list.

When implementing the SvS algorithm over uncompressed integers, it is convenient to be able to write out the result of the intersection directly in one of the two lists. This removes the need to allocate additional memory to save the result of the intersection. All our intersection algorithms are designed to have the *output-to-input* property: it is always safe to write the result in the shorter of the two lists, because the algorithms never overwrite unprocessed data. The algorithms by Schlegel et al. and Katsov do not have this property.

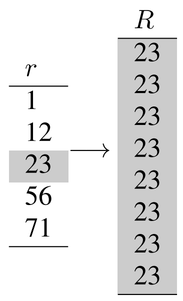
Our simplest SIMD intersection algorithm is V1 (see Algorithm 2): V1 stands for “first vectorized intersection algorithm”. It is essentially equivalent to a simple textbook intersection algorithm (SCALAR) except that we access the integers of the long lists in blocks of T integers. We advance in the short list one integer at a time. We then advance in the long list until we find a block of T integers such that the last one is at least as large as the integer in the short list. We compare the block of T different integers in the long list with the integer in the short list using SIMD instructions. If there is a match, that is, if one of the T integers is equal to the integer from the short list, we append that integer to the intersection list.

For example, consider Fig. 4. In the short list r, we want to know whether the integer 23 is present in the longer list. We start by creating a vector containing $T = 8$ copies of the integer: $R = (23, 23, \ldots, 23)$. From the long list, we load 8 consecutive integers such that the last integer in the list is at least as large as 23. In this case, we load $F = (15, 16, \ldots, 29, 31)$. Then we compare the vectors R and F using as little as two SIMD operations. In our example, there is a match. There is no need to do much further work: we have determined that the integer r belongs to the intersection.

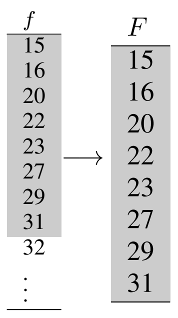
Algorithm 2 The V1 intersection algorithm

```
Require: SIMD architecture with T-integer vectors
 1: input: two sorted non-empty arrays of integers $r, f$
 2: assume: $\text{length}(f)$ is divisible by T
 3: $x$ initially empty dynamic array (our answer)
 4: $j \leftarrow 1$
 5: for $i \in \{1, 2, \ldots, \text{length}(r)\}$ do
 6:     $R \leftarrow (r_i, r_i, \ldots, r_i)$
 7:     while $f_{j-1+\mathrm{T}} < r_i$ do
 8:         $j \leftarrow j + \mathrm{T}$
 9:         if $j > \text{length}(f)$ then
10:             return $x$
11:     $F \leftarrow (f_j, f_{j+1}, \ldots, f_{j-1+\mathrm{T}})$
12:     if $R_i = F_i$ for some $i \in \{1, 2, \ldots, \mathrm{T}\}$ then
13:         append $r_i$ to $x$
14: return $x$
```

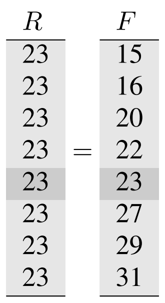
We found it best to use the V1 algorithm with $\mathrm{T} = 8$. Because SSE2 only offers 4-integer vectors, we simulate an 8-integer vector by using two 4-integer vectors (see Appendix A). To compare two vectors for possible equality between any two components, we use the `pcmpeqd` instruction: given



(a) Extract one value r_i from r and create T-integer vector filled with this one value $R = (r_i, r_i, \ldots)$ (e.g., with the `pshufd` SSE2 instruction)



(b) Load T values from f in vector F (e.g., using the `movdqu` SSE2 instruction)



(c) Determine whether $R_i = F_i$ for some i (e.g., using the `pcmpeqd` and `movmskps` instructions)

Figure 4. Brief illustration of Algorithm V1 with $\mathrm{T} = 8$ and $r_i = 23$

4 pairs of 32-bit integers, it generates 4 integers: `0xFFFFFFFF` when the two integers are equal and `0x00000000` otherwise. We can combine the result of two such tests with a bitwise OR (`por`). To test whether there is a match of any of the integers, we use the `movmskps` instruction.

However, the computational complexity of the V1 algorithm is still $O(m + n/\mathrm{T})$ where T is limited to small constants by the CPU architecture. Hence, a simple galloping intersection can be faster than V1 when $n \gg m$ as the galloping algorithm has complexity $O(m \log n)$.

We can optimize further and add two layers of branching (henceforth V3, see Algorithm 3). That is, like with V1, we first locate a block of $4T$ integers $(f_j, \ldots, f_{j-1+4T})$ in the larger array f where a match could be found for the current value r_i selected from the small array: we increment j by steps of $4T$ until the last value from the current block is at least as large as the value from the small array, $f_{j-1+4T} \geq r_i$. However, instead of directly comparing $4T$ pairs of integers as the analog of Algorithm V1 would do, we use the binary search algorithm to find the one block of T integers within the larger block of $4T$ integers where a match is possible. Technically, we compare the current value from the small array r_i with the value in the middle of the block of $4T$ (integers f_{j-1+2T}) and then again with either the value in the middle of the first half (f_{j-1+T}) or in the middle of the second half (f_{j-1+3T}). Thus, some comparisons can be skipped. We first tried a version, called V2, that added only one layer of branching to V1 but it did not prove useful in practice.

However, when n is large, galloping is superior to both V1 and V3. Thus, we also created a SIMD-based galloping (henceforth SIMD GALLOPING, see Algorithm 4). It uses the same ideas as galloping search, except that we exploit the fact that SIMD instructions can compare T pairs of integers at once. SIMD GALLOPING has similar complexity in terms of m and n as scalar galloping ($O(\frac{m}{\mathrm{T}} \log n) = O(m \log n)$) so we expect good scalability.

6.2.1. SIMD Hybrid Algorithm In practice, we find that our SIMD GALLOPING is always faster than a non-SIMD galloping implementation. Nevertheless, to fully exploit the speed of SIMD instructions, we find it desirable to still use V1 and V3 when they are faster. Thus, when processing 32-bit integers, we use a combination of V1, V3, and SIMD galloping where a choice of the intersection algorithm is defined by the following heuristic:

- When $\mathrm{length}(r) \leq \mathrm{length}(f) < 50 \times \mathrm{length}(r)$, we use the V1 algorithm with 8-integer vectors ($\mathrm{T} = 8$, see Algorithm 2).
- When $50 \times \mathrm{length}(r) \leq \mathrm{length}(f) < 1000 \times \mathrm{length}(r)$, we use the V3 algorithm with 32-integer vectors ($\mathrm{T} = 32$, see Algorithm 3)
- When $1000 \times \mathrm{length}(r) \leq \mathrm{length}(f)$, we use the SIMD GALLOPING algorithm with 32-integer vectors ($\mathrm{T} = 32$, see Algorithm 4).

Though we are most interested in 32-bit integers, V1, V3 and SIMD GALLOPING can also be used with arrays of 8-bit, 16-bit and 64-bit integers. Different heuristics would be needed in such cases.

6.2.2. Possible refinements Our SIMD Intersection algorithms assume that the longer array has length divisible by either T or 4T. In practice, when this is not the case, we complete the computation of the intersection with the SCALAR algorithm. We expect that the contribution of this step to the total running time is small. As a possibly faster alternative, we could pad the longer array to a desired length using a special integer value that is guaranteed not to be in the shorter array. We could also fallback on other SIMD algorithms to complete the computation of the intersection.

We could introduce other refinements. For example, our implementation of SIMD GALLOPING relies on a scalar binary search like the conventional galloping. However, recent Intel processors have introduced gather instructions (`vpgatherdd`) that can retrieve at once several integers from various locations [33]. Such instructions could accelerate binary search. We do not consider such possibilities further.

7. EXPERIMENTAL RESULTS

We assess experimentally the unpacking speed with *integrated* differential coding (§ 7.4), the decompression speed of the corresponding schemes (§ 7.5), the benefits of our SIMD-based intersection schemes (§ 7.6), and, SIMD-accelerated bitmap-list hybrids (§ 7.7) with realistic query logs and inverted indexes.

7.1. Software

All our software is freely available online[‡] under the Apache Software License 2.0. The code is written in C++ using the C++11 standard. Our code builds using several compilers such as `clang++` 3.2 and Intel `icpc` 13. However, we use `GNU GCC` 4.7 on a Linux PC for our tests. All code was compiled using the `-O3` flag. We implemented scalar schemes (FASTPFOR and VARINT) without using SIMD instructions and with the scalar equivalent of D1 differential coding.

7.2. Hardware

We ran all our experiments on an Intel Xeon CPU (E5-1620, *Sandy Bridge*) running at 3.60 GHz. This CPU also has 10 MB of L3 cache as well as 32 kB and 256 kB of L1 and L2 data cache per core. We have 32 GB of RAM (DDR3-1600) running quad-channel. We estimate that we can read from RAM at a speed of 4 billion 32-bit integers per second and from L3 cache at a speed of 8 billion 32-bit integers per second. All data is stored in RAM so that disk performance is irrelevant.

[‡]`https://github.com/lemire/SIMDCompressionAndIntersection`.

Algorithm 3 The V3 intersection algorithm

```
Require: SIMD architecture with T-integer vectors
1: input: two sorted non-empty arrays of integers $r, f$
2: assume: $\mathrm{length}(f)$ is divisible by $4\mathrm{T}$
3: $x$ initially empty dynamic array (our answer)
4: $j \leftarrow 1$
5: for $i \in \{1, 2, \ldots, \mathrm{length}(r)\}$ do
6:     $R \leftarrow (r_i, r_i, \ldots, r_i)$
7:     while $f_{j-1+4\mathrm{T}} < r_i$ do
8:         $j \leftarrow j + 4\mathrm{T}$
9:         if $j > \mathrm{length}(f)$ then
10:            return $x$
11:    if $f_{j-1+2\mathrm{T}} \geq r_i$ then
12:        if $f_{j-1+\mathrm{T}} \geq r_i$ then
13:            $F \leftarrow (f_j, f_{j+1}, \ldots, f_{j-1+\mathrm{T}})$
14:        else
15:            $F \leftarrow (f_{j+\mathrm{T}}, f_{j+2+\mathrm{T}}, \ldots, f_{j-1+2\mathrm{T}})$
16:    else
17:        if $f_{j-1+3\mathrm{T}} \geq r_i$ then
18:            $F \leftarrow (f_{j+2\mathrm{T}}, f_{j+2+2\mathrm{T}}, \ldots, f_{j-1+3\mathrm{T}})$
19:        else
20:            $F \leftarrow (f_{j+3\mathrm{T}}, f_{j+2+3\mathrm{T}}, \ldots, f_{j-1+4\mathrm{T}})$
21:    if $R_i = F_i$ for some $i \in \{1, 2, \ldots, \mathrm{T}\}$ then
22:        append $r_i$ to $x$
23: return $x$
```

Algorithm 4 The SIMD GALLOPING algorithm

```
Require: SIMD architecture with T-integer vectors
1: input: two non-empty sorted arrays of integers $r, f$
2: assume: $\mathrm{length}(f)$ is divisible by $\mathrm{T}$
3: $x$ initially empty dynamic array (our answer)
4: $j \leftarrow 1$
5: for $i \in \{1, 2, \ldots, \mathrm{length}(r)\}$ do
6:     $R \leftarrow (r_i, r_i, \ldots, r_i)$
7:     find by sequential search the smallest $\delta$ such that $f_{j+\delta-1+\mathrm{T}} \geq r_i$ for $\delta = 0, 1\mathrm{T}, 2\mathrm{T}, 4\mathrm{T}, \ldots, \mathrm{length}(f) - 1 + \mathrm{T}$, if none return $x$
8:     find by binary search the smallest $\delta_{\min}$ in $[\lfloor \delta/2 \rfloor, \delta]$ divisible by $\mathrm{T}$ such that $f_{j+\delta_{\min}-1+\mathrm{T}} \geq r_i$
9:     $j \leftarrow j + \delta_{\min}$
10:    $F \leftarrow (f_j, f_{j+1}, \ldots, f_{j-1+\mathrm{T}})$
11:    if $R_i = F_i$ for some $i \in \{1, 2, \ldots, \mathrm{T}\}$ then
12:        append $r_i$ to $x$
13: return $x$
```

7.3. Real data

To fully assess our results, we need realistic data sets. For this purpose, we use posting lists extracted from the TREC collections ClueWeb09 (Category B) and GOV2. Collections GOV2 and ClueWeb09 (Category B) contain roughly 25 and 50 million HTML documents, respectively. These documents were crawled from the web. In the case of GOV2 almost all pages were collected from websites in either the `.gov` or `.us` domains, but the ClueWeb09 crawl was not limited to any specific domain.

ClueWeb09 is a partially sorted collection: on average it has runs of 3.7 thousand documents sorted by URLs. In GOV2, the average length of the sorted run is almost one. Thus, we use two variants of GOV2: the original and a sorted one, where documents are sorted by their URLs. We did not consider other sorting strategies such as by number of terms in the document (terms-in-document) or by document size [15, 34].

We first indexed collections using Lucene (version 4.6.0): the words were stopped using the default Lucene settings, but not stemmed. Then, we extracted postings corresponding to one million most frequent terms. In both GOV2 and ClueWeb09, the excluded terms represent $4\,\%$ of the postings. Morever, the excluded posting lists had averages of 3.2 postings for GOV2 and 4.5 postings for ClueWeb09. The extracting software as well as the processed data is freely available online.[§] Uncompressed, extracted posting lists from GOV2 and ClueWeb09 use 23 GB and 59 GB, respectively. They include only document identifiers.

Our corpora represent realistic sets of documents obtained by splitting a large collection so that each part fits into the memory of a single server. In comparison, other researchers use collections of similar or smaller sizes. Culpepper and Moffat [5], Ao et al. [17], Barbay et al. [14], Ding et al. [16] and Vigna [19] used TREC GOV2 (25M documents), Ding and König [9] indexed Wikipedia (14M documents) while Transier and Sanders [10] used WT2g (250k documents).

7.3.1. Query logs In addition, we used the TREC million-query track (1MQ) logs (60 thousand queries from years 2007–2009). We randomly picked 20 thousand queries containing at least two indexed terms. These queries were converted into sequences of posting identifiers for GOV2 and ClueWeb09. Recall that postings beyond one million most frequent terms were eliminated. Therefore, we had to exclude $4.6\,\%$ of queries in the case of ClueWeb09 and $8.4\,\%$ of queries in the case of GOV2 (which could not be converted to posting identifiers). We believe that excluding a small fraction of the queries slightly biased up processing times, but the overall effect was not substantial.

Table IV gives intersection statistics for the GOV2 and Clueweb09 collections. We give the percentage of all queries having a given number of terms, the average size of the intersection, along with the average size of the smallest posting list (term 1), the average size of the second smallest (term 2) and so on.

To insure that all our indexes fit in RAM, we index only the terms used by the given query log. When reporting compression results (in bits/int), we only consider this in-memory index.

7.4. Bit unpacking

Bit unpacking (see § 4.2) is the fundamental operation we use as building blocks for our compression schemes (i.e., the families S4-BP128, FastPFOR, S4-FastPFOR). In our experiments, unpacking always includes differential coding, as an integrated operation or a separate step (see § 5).

To test the speed of our various bit unpacking routines, we generated random arrays of 4096 32-bit integers for each bit width $b = 1, 2, \ldots, 31$. We generated gaps smaller than 2^b using the C `rand` function as a pseudo-random number generator. Because SIMD unpacking routines operate on blocks of 128 integers (see Algorithm 1), a total of $4096/128 = 32$ calls to the same unpacking routine is required to unpack an entire array. For each bit width, we ran 2^{14} sequences of packing and unpacking and report only the average.

Generally, unpacking is fastest for low bit widths and for bit widths that are powers of two. We used the Intel Architecture Code Analyzer (IACA) on our 32 SIMD unpacking routines—omitting differential coding for simplicity. IACA provides an optimistic estimation of the reciprocal throughput of these procedures: a value of 32 cycles means that if we repeatedly called the same routine, one full routine might complete every 32 cycles. Thus, in our case, we get an optimistic estimate of the number of cycles it takes to unpack 128 integers. Fig. 5 provides this reciprocal

[§]`https://github.com/searchivarius/IndexTextCollect` and `http://lemire.me/data/integercompression2014.html`.

Table IV. Statistics about the TREC Million-Query log on two corpora. We give the percentage of all queries having a given number of terms, the average size of the intersection along with the average size of the smallest posting list (term 1), the average size of the second smallest (term 2) and so on. All sizes are in thousands. There are 50M documents in total for Clueweb09 and half as many for Gov2.

(a) Clueweb09 (matching documents in thousands)

#	%	inter.	term 1, term 2, . . .
2	19.8	93	380, 2600
3	32.5	29	400, 1500, 5100
4	26.3	17	480, 1400, 3200, 8100
5	13.2	12	420, 1200, 2600, 4800, 10 000
6	4.9	4	350, 1000, 2100, 3700, 6500, 13 000
7	1.7	5	390, 1100, 2100, 3400, 5200, 7300, 13 000

(b) Gov2 (matching documents in thousands)

#	%	inter.	term 1, term 2 , . . .
2	19.6	49	160, 1100
3	32.3	22	180, 710, 2400
4	26.4	11	210, 620, 1500, 3700
5	13.4	7	170, 520, 1100, 2200, 4400
6	5.0	3	140, 420, 850, 1500, 2600, 5100
7	1.8	9	190, 440, 790, 1300, 2200, 3200, 5400

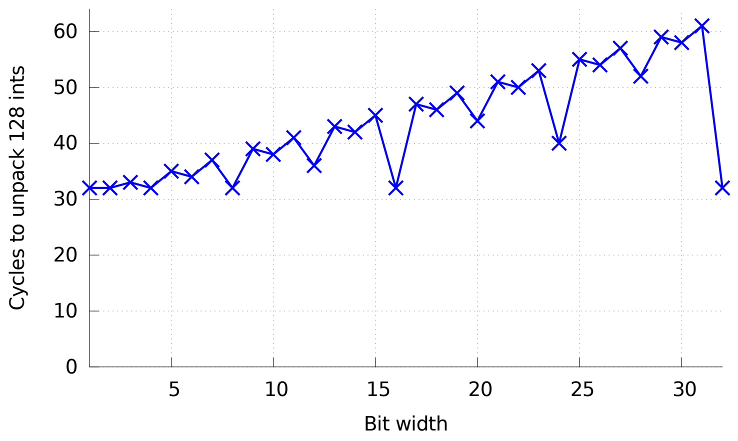
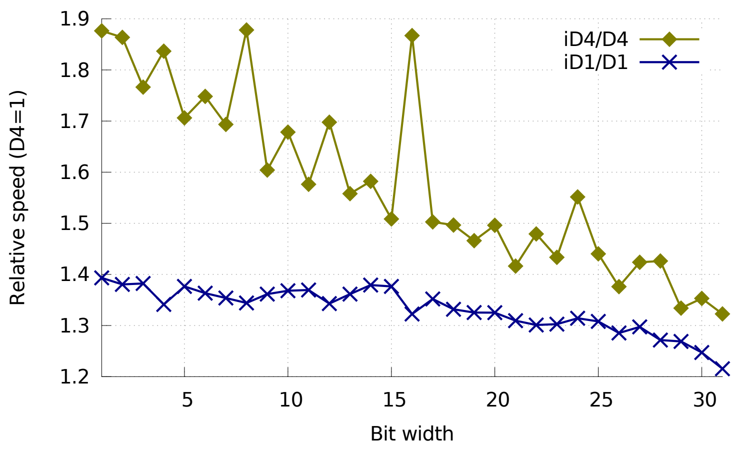


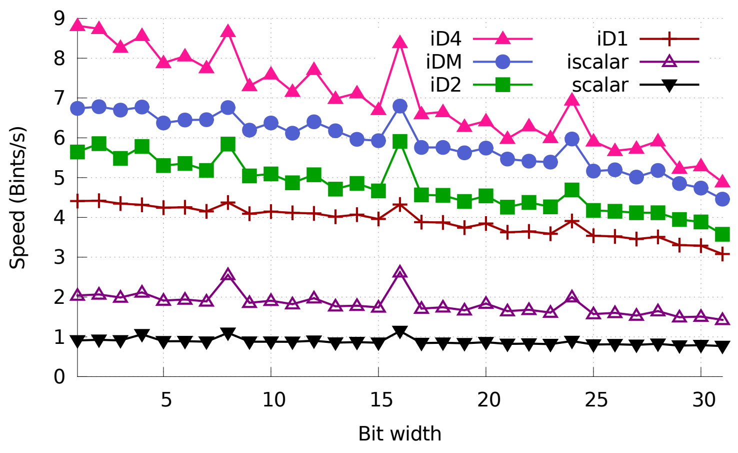
Figure 5. Estimated reciprocal throughput of the SIMD unpacking procedures according to IACA (for 128 32-bit integers and without differential coding).

throughput for all bit widths on our processor microarchitecture (*Sandy Bridge*). It takes between 32 to 61 cycles to execute each 128-integer unpacking procedure, and the estimated throughput depends on the bit width. IACA reports that execution port 4 is a bottleneck for bit widths 1, 2, 3, 4, 8, 16, 32: it indicates that we are limited by register stores in these cases. Indeed, 32 128-bit store operations are required to write 128 32-bit integers and we can only issue one store instruction per cycle: we therefore need 32 cycles. Otherwise, execution port 0 is a bottleneck. This port is used by shift instructions on *Sandy Bridge*: these instructions are used more often when bit widths are large, except if the bit width is a power of two.

In Fig. 6a, we plot the speed ratios of the integrated bit unpacking (i.e., Algorithm 1 for SIMD routines), vs. the regular one where differential coding is applied separately on small blocks of 32-bit integers. To differentiate the integrated and regular differential coding we use an i prefix (iD4 vs. D4). For D4, integration improves the speed by anywhere from over 30 % to 90 %. For D1, the



(a) One-pass (integrated) vs. two-pass (block-level) prefix sum



(b) Several differential bit unpacking speeds

Figure 6. Unpacking speed for all bit widths ($b = 1, \ldots, 31$). See Algorithm 1. Speed is reported in billions of 32-bit integers per second. We use arrays of 4096 integers for these tests.

results are less impressive as the gains range from 20 % to 40 %. For D2 and DM, the result lies in-between. Moreover, the gains due to the integration are most important when the unpacking is fastest: for small bit widths or bit widths that are a power of two such as 8 and 16. This phenomenon is consistent with our analysis (see § 5): the benefits of integration are mostly due to the reduction by half of the number of registers stored to memory, and these stores are more of a bottleneck for these particular bit widths.

In Fig. 6b, we compare all of the integrated bit unpacking procedures (iD4, iDM, iD2, iD1). As our analysis predicted (see Table III), D4 is the fastest followed by DM, D2 and D1. For comparison, we also include the scalar bit unpacking speed. The integrated bit unpacking (iscalar curve) is sometimes twice as fast as regular bit unpacking (scalar curve). Even so, the fastest scalar unpacking routine has half the speed as our slowest SIMD unpacking routine (iD1).

7.5. Decompression speed

We consider several fast compression schemes: VARINT (see § 4.1), S4-BP128 (see § 4.3), FASTPFOR and S4-FASTPFOR (see § 4.4). To test the speed of our integer compression schemes, we generated arrays using the ClusterData distribution from Anh and Moffat [25]. This distribution primarily leaves small gaps between successive integers, punctuated by occasional larger gaps. We

Table V. Results on ClusterData for dense (2^{16} integers in $[0, 2^{19})$) and sparse (2^{16} integers in $[0, 2^{30})$). We report decompression speed in billions of 32-bit integers per second (Bint32/s). Shannon entropy of the deltas is given. Our Intel CPU runs at 3.6 GHz. The -NI suffix (for *non-integrated*) indicates that the prefix sum requires a second pass over 128-integer blocks. The standard deviation of our timings is less than 5 %.

	dense		sparse	
	bits/int	Bint32/s	bits/int	Bint32/s
entropy	3.9	–	14.7	–
copy	32.0	5.4	32.0	5.4
S4-BP128-D4	6.0	5.4	16.5	4.4
S4-BP128-D4-NI	6.0	3.9	16.5	3.3
S4-BP128-DM	5.9	5.5	16.3	4.1
S4-BP128-DM-NI	5.9	3.9	16.3	3.3
S4-BP128-D2	5.5	4.8	16.0	3.5
S4-BP128-D2-NI	5.5	3.7	16.0	3.2
S4-BP128-D1	5.0	3.9	15.5	3.0
S4-BP128-D1-NI	5.0	3.0	15.5	2.7
S4-FastPFOR-D4	5.8	3.1	16.1	2.6
S4-FastPFOR-DM	5.5	2.8	15.8	2.4
S4-FastPFOR-D2	5.1	2.7	15.4	2.4
S4-FastPFOR-D1	4.4	2.2	14.8	2.0
FastPFOR	4.4	1.1	14.8	1.1
varint	8.0	1.2	17.2	0.3

generated arrays of 65536 32-bit integers in either $[0, 2^{19})$ or $[0, 2^{30})$. In each case, we give the decompression speed of our schemes, the entropy of the deltas as well as the speed (in billions of 32-bit integers per second) of a simple copy (implemented as a call to `memcpy`) in Table V. All compression schemes use differential coding. We append a suffix (-D4, -DM, -D2, and -D1) to indicate the type of differential coding used. We append the -NI suffix (for *non-integrated*) to S4-BP128-* schemes where the prefix sum requires a second pass. We get our best speeds with S4-BP128-D4 and S4-BP128-DM.¶ Given our 3.60 GHz clock speed, they decompress 32-bit integers using between 0.7 and 0.8 CPU cycles per integer. In contrast, the best results reported by Lemire and Boytsov [3] was 1.2 CPU cycles per 32-bit integer, using similar synthetic data with a CPU of the same family (*Sandy Bridge*).

S4-BP128-D1-NI is 38 % faster than S4-FastPFOR but S4-FastPFOR compensates with a superior compression ratio (5 %–15 % better). However, with integration, S4-BP128-D1 increases the speed gap to 50 %–75 %.

7.6. *Intersections*

We present several intersections algorithms between (uncompressed) lists of 32-bit integers in § 6. To test these intersections, we again generate lists using the ClusterData distribution [25]: all lists contain integers in $[0, 2^{26})$. Lists are generated in pairs. First, we set the target cardinality n of the largest lists to 2^{22}. Then we vary the target cardinality m of the smallest list from n down to $n/10000$. To ensure that intersections are not trivial, we first generate an "intersection" list of size $m/3$ (rounded to the nearest integer). The smallest list is built from the union of the intersection list with another list made of $2m/3$ integers. Thus, the maximum length of this union is m. The longest list is made of the union of the intersection list with another list of cardinality $n - m/3$, for a total cardinality of up to n. The net result is that we have two sets with cardinalities $\approx m$ and $\approx n$ having an intersection at least as large as $m/3$. We compute the average of 5 intersections

¶Though we report a greater speed in one instance for S4-BP128-DM (5.5) than for S4-BP128-D4 (5.4), the 3-digit values are close: 5.45 and 5.44.

Table VI. Time required to answer queries in ms/query along with the storage requirement in bits/int, using the TREC Million-Query log sample from § 7.3.1. The standard deviation of our timings is less than 1 %.

scheme	bits/int	time	bits/int	time	bits/int	time
	GOV2 (sorted)		GOV2 (unsorted)		CLUEWEB09	
SIMD SvS	32.0	0.5	32.0	0.7	32.0	1.5
Galloping SvS	32.0	0.7	32.0	1.4	32.0	2.8
SCALAR SvS	32.0	2.8	32.0	3.3	32.0	6.6

(using 5 pairs of lists). This choice ($m/3$) is motivated by our experience with search engine queries (e.g., see Table IV) where the intersection size is often about 30 % of the smaller of two sets for 2-term queries. We also present the figures for the case where the intersection is much smaller ($0.01m$): the relative results are similar. The performance of the intersection procedures is sensitive to the data distribution however. When we replaced ClusterData with a uniform distribution, the intersection speed diminished by up to a factor of 2.5: value clustering improves branch predictions and skipping [27].

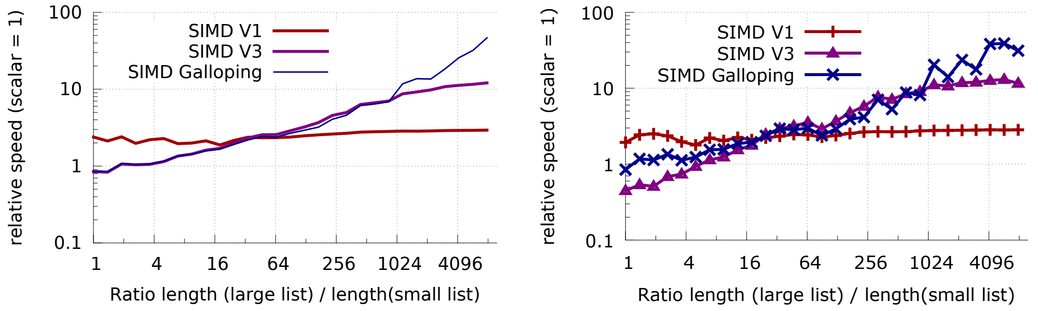
We report all speeds relative to the basic SCALAR intersection. All input lists are uncompressed. In Fig. 7a, we compare the speed of our 3 SIMD intersection functions: V1, V3, and SIMD GALLOPING. We see that V1 is the fastest for ratios of up to 16:1, whereas SIMD GALLOPING is the fastest for ratios larger than 1024:1. In-between, V3 is sometimes best. This justifies our heuristic which uses V1 for ratios up to 50:1, V3 for ratios up to 1000:1, and SIMD GALLOPING for larger ratios. In Fig. 7b, we compare the two non-SIMD intersections (SCALAR and galloping) with our SIMD intersection procedure. We see that for a wide range of cases (up to a ratio of 64:1), our SIMD intersection procedure is clearly superior to the scalar galloping, being up to twice as fast. As the ratio increases, reflecting a greater difference in list sizes, non-SIMD galloping eventually becomes nearly as fast as our SIMD intersection. We also compare with Katsov's algorithm (see Fig. 7b). Katsov is only competitive with our algorithms when the lists have similar lengths.

We also included our versions of Schlegel et al.'s algorithm: the original and an improved version relying on the `pcmpistrm` instruction (see Fig. 7c). In these particular tests, the 32-bit integers are first transformed into Schlegel et al.'s specialized data structure, as several arrays of 16-bit integers. We also output the answer in this format. The improved version of Schlegel et al.'s algorithm is about 15 % faster, in the best case. This supports our claim that the `pcmpistrm` instruction is preferable to the `pcmpestrm` instruction. We compared Schlegel et al. with a version of V1 adapted to lists of 16-bit integers. Like the 32-bit V1, the 16-bit V1 compares blocks of 256 bits and thus compares sixteen pairs of 16-bit integers at a time (i.e., $T = 16$). We find that Schlegel et al. is better than V1 as long as the ratio of lengths is less than 10. When one list is much longer than the other one, V1 becomes much faster (up to 75× faster in this test). We do not use Schlegel et al.'s 16-bit format further, but our results suggest that we could combine V1 and Schlegel et al.'s algorithm for best results. That is, we could use Schlegel et al. for lists having similar lengths, and V1 otherwise.

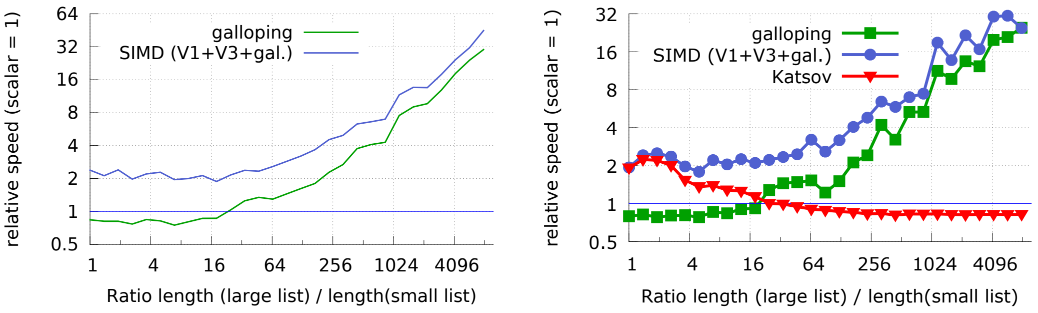
We also evaluated our algorithms using postings lists for our three collections using our TREC 1M Query log (see Table VI and Fig. 8). For this test, posting lists are uncompressed. We see that our SIMD intersection routine is nearly twice as fast as (non-SIMD) galloping. In the sorted version of Gov2, we expect most gaps between document identifiers to be small, with a few large gaps. In contrast, the unsorted version of Gov2 has a more uniform distribution of gaps. Just as we find that intersections are faster on the ClusterData distributions, we find that intersections are 1.4–2× faster on the sorted Gov2 vs. its unsorted counterpart. In the next section (§ 7.7), we show that a much better speed is possible if we use bitmaps to represent some posting lists.

7.7. *Bitmap-list hybrids (*HYB+M2*)*

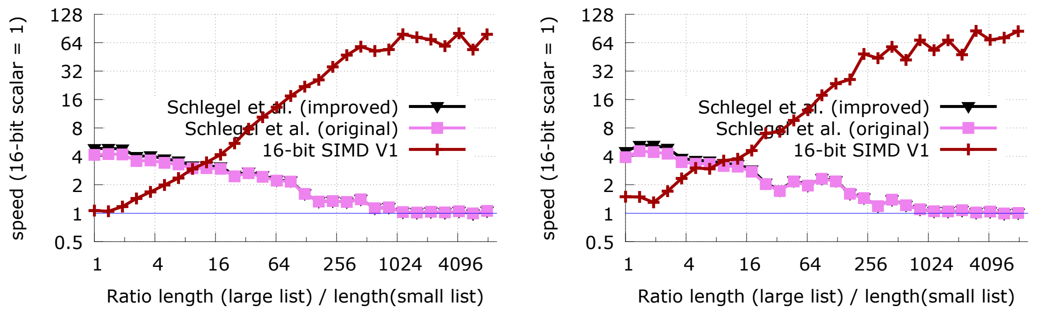
We can also use bitmaps to accelerate intersections. Culpepper and Moffat's HYB+M2 framework is simple but effective [5]: all lists where the average gap size is smaller or equal to B (where $B = 8, 16$ or 32) are stored as bitmaps whereas other lists are stored as compressed deltas (using



(a) SIMD V1 vs. SIMD V3 vs. SIMD Galloping ($m/3$ left, $0.01m$ right)



(b) SIMD vs. non-SIMD galloping Intersection vs. Katsov ($m/3$ left, $0.01m$ right)



(c) 16-bit case ($m/3$ left, $0.01m$ right)

Figure 7. Intersection between two lists of different cardinality as described in § 7.6. We provide both the case where the size of the intersection is $m/3$ and where the size of the intersection is $0.01m$.

VARINT or another compression algorithm). To compute any intersection, the compressed lists are first intersected (e.g., using galloping) and the result is then checked against the bitmaps one by one. It is a hybrid approach in the sense that it uses both bitmaps and compressed deltas. We modify their framework by replacing the compression and intersection functions with SIMD-based ones. To simplify our analysis, we use galloping intersections with scalar compression schemes such as VARINT and FASTPFOR, and SIMD intersections with SIMD compression schemes. Intersections between lists are always computed over uncompressed lists. SIMD intersections are computed using the hybrid made of V1, V3 and SIMD GALLOPING presented in § 6.2.1.

To improve data cache utilization, we split our corpora into parts processed separately (32 parts for GOV2 and 64 parts for ClueWeb09). The partition is based on document identifiers: we have, effectively, 32 or 64 independent indexes that apply to different document subsets. Thus, all intermediate results for a single part can fit into L3 cache. Given a query, we first obtain the result for the first part, then for the second, and so on. Finally, all partial results are collected in one array. In each part, we work directly on compressed lists and bitmaps, without other auxiliary data structures.

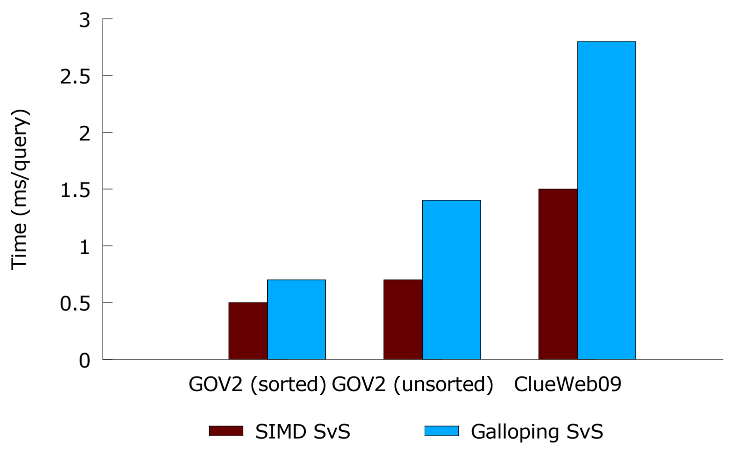


Figure 8. Average time required to answer queries over uncompressed posting lists.

The lists are decompressed and then intersected. We expect our approach to be reasonably efficient given our benchmark: in Appendix B, we validate it against an alternative that involves skipping.

As shown in Table VII and Fig. 9, the schemes using SIMD instructions are 2–3 times faster than FASTPFOR and VARINT while having comparable compression ratios. The retrieval times can be improved (up to $\approx 4\times$) with the addition of bitmaps, but the index is sometimes larger (up to $\approx 2\times$).

For $B = 8$, $B = 16$, and indices without bitmaps, the various S4-BP128-* schemes provide different space vs. performance trade offs. For example, for ClueWeb09 and $B = 8$, S4-BP128-D4 is 11 % faster than S4-BP128-D1, but S4-BP128-D1 uses 7 % less space.

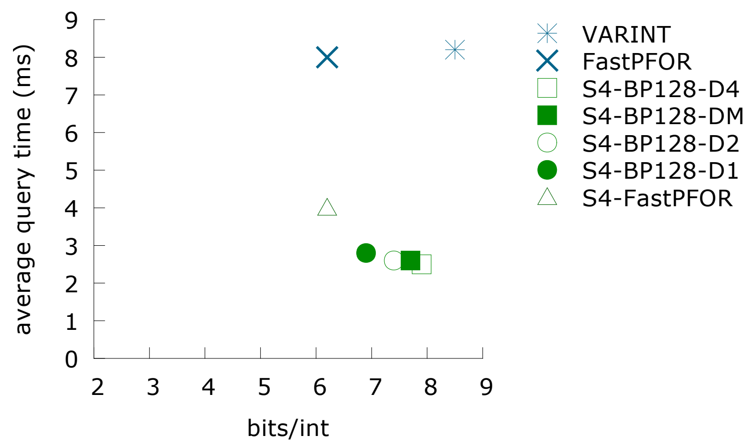
We also compare scalar vs. SIMD-based algorithms. For ClueWeb09, S4-FASTPFOR-D1 can be nearly twice as fast as the non-SIMD FASTPFOR scheme (see Fig. 10). Even with $B = 16$ where we rely more on the bitmaps for speed, S4-FASTPFOR is over 50 % faster than FASTPFOR (for ClueWeb09). As expected, for $B = 32$, there is virtually no difference in speed and compression ratios among various S4-BP128-* compression schemes. However, S4-BP128-D4 is still about twice as fast as VARINT so that, even in this extreme case, our SIMD algorithms are worthwhile.

As a reference point, we also implemented intersections using the Skipper [35] approach. In this last case, posting lists are stored as deltas compressed using VARINT. However, to support random access, we have an auxiliary data structure containing sampled uncompressed values with pointers inside the array of compressed deltas. We found that sampling every 32 integers gave good results. Because we need to store both the uncompressed integer and a pointer for each 32 integers, the auxiliary Skipper data structure adds about two bits of storage per integer. We refer to the original paper and to our software for details. We find that Skipper was 2 to 3 times faster than intersections over VARINT-compressed deltas (at the expense of storage) without bitmap, but compared with the HYB+M2 framework ($B = 8, 16, 32$) or our SIMD-based approaches, it is not competitive. Though we could replace VARINT in Skipper by a faster alternative, we expect that the benefits of skipping would then be diminished.

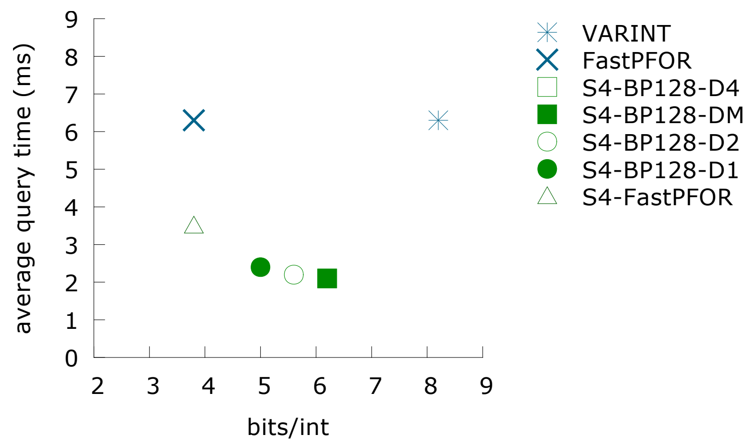
We also recorded the median and the 90th percentile retrieval times. All these metrics benefited from our techniques. Moreover, our measures are accurate: after repeating experiments five times, we find that all standard deviations are less than 1 %.

8. DISCUSSION AND CONCLUSIONS

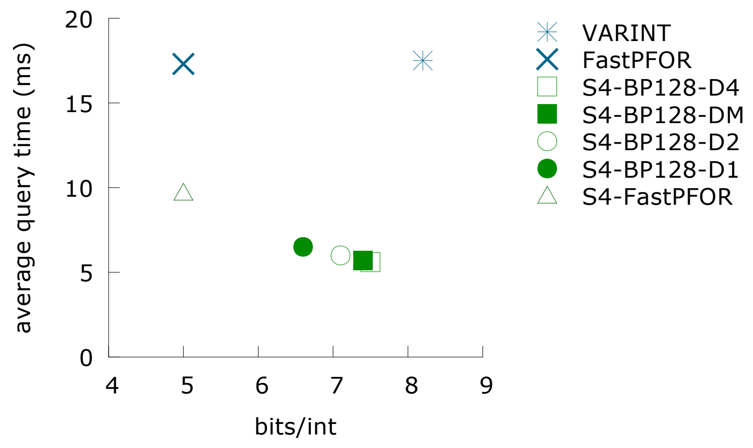
We demonstrated that combining unpacking and differential coding resulted in faster decompression speeds, which were approximately 30 % better than the best speeds reported previously [3]. To match the performance of these fast compression schemes, we additionally vectorized and optimized the intersection of posting lists. To this end, we introduced a family of algorithms exploiting



(a) GOV2 (unsorted)



(b) GOV2 (sorted)



(c) CLUEWEB09

Figure 9. Average time required to answer queries vs. the storage requirement in bits/int without bitmap

commonly available SIMD instructions (V1, V3 and SIMD GALLOPING). They are often twice as fast as the best non-SIMD algorithms. Then, we used our fast SIMD routines for decompression and

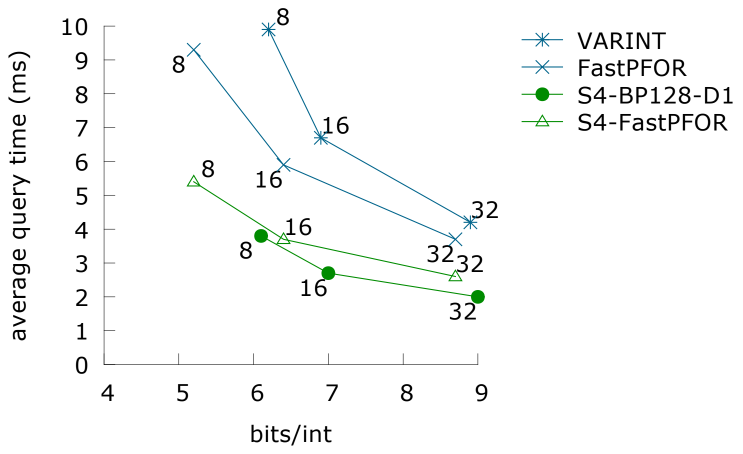


Figure 10. Average time required to answer queries vs. the storage requirement in bits/int using partitions and various bitmap parameters ($B = 8, 16, 32$) for CLUEWEB09.

posting intersection to accelerate Culpepper and Moffat's HYB+M2 [5]. We believe that HYB+M2 is one of the fastest published algorithms for conjunctive queries. Yet we were able to sometimes double the speed of HYB+M2 without sacrificing compression.

Our work was focused on 128-bit vectors. Intel and AMD recently released processors that support integer operations on 256-bit vectors using the new AVX2 instruction set. On such a processor, Willhalm et al. [36] were able to double their bit unpacking speed. Moreover, Intel plans to support 512-bit vectors in 2015 on its commodity processors. Thus optimizing algorithms for SIMD instructions will become even more important in the near future.

REFERENCES

1. Catena M, Macdonald C, Ounis I. On inverted index compression for search engine efficiency. *Advances in Information Retrieval*, *Lecture Notes in Computer Science*, vol. 8416. Springer International Publishing, 2014; 359–371, doi:10.1007/978-3-319-06028-6_30.
2. Büttcher S, Clarke CLA. Index compression is good, especially for random access. *Proceedings of the 16th ACM conference on Information and Knowledge Management*, CIKM '07, ACM: New York, NY, USA, 2007; 761–770, doi:10.1145/1321440.1321546.
3. Lemire D, Boytsov L. Decoding billions of integers per second through vectorization. *Softw. Pract. Exper.* 2015; **45**(1), doi:10.1002/spe.2203.
4. Ladner RE, Fischer MJ. Parallel prefix computation. *J. ACM* Oct 1980; **27**(4):831–838, doi:10.1145/322217.322232.
5. Culpepper JS, Moffat A. Efficient set intersection for inverted indexing. *ACM Trans. Inf. Syst.* Dec 2010; **29**(1):1:1–1:25, doi:10.1145/1877766.1877767.
6. Tatikonda S, Cambazoglu BB, Junqueira FP. Posting list intersection on multicore architectures. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, SIGIR '11, ACM: New York, NY, USA, 2011; 963–972, doi:10.1145/2009916.2010045.
7. Kim Y, Seo J, Croft WB. Automatic boolean query suggestion for professional search. *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, ACM: New York, NY, USA, 2011; 825–834, doi:10.1145/2009916.2010026.
8. Stepanov AA, Gangolli AR, Rose DE, Ernst RJ, Oberoi PS. SIMD-based decoding of posting lists. *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, ACM: New York, NY, USA, 2011; 317–326, doi:10.1145/2063576.2063627.
9. Ding B, König AC. Fast set intersection in memory. *Proc. VLDB Endow.* 2011; **4**(4):255–266.
10. Transier F, Sanders P. Engineering basic algorithms of an in-memory text search engine. *ACM Trans. Inf. Syst.* Dec 2010; **29**(1):2:1–2:37, doi:10.1145/1877766.1877768.
11. Moffat A, Zobel J. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.* 1996; **14**(4):349–379.
12. Baeza-Yates R. A fast set intersection algorithm for sorted sequences. *Combinatorial Pattern Matching*, *LNCS*, vol. 3109, Sahinalp S, Muthukrishnan S, Dogrusoz U (eds.). Springer Berlin / Heidelberg, 2004; 400–408.
13. Demaine ED, López-Ortiz A, Munro JI. Adaptive set intersections, unions, and differences. *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '00, Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2000; 743–752.

Table VII. Time required to answer queries from TREC 1MQ in ms/query along with the storage requirement in bits/int. Entropies of the deltas are provided. We write S4-FASTPFOR as a shorthand for S4-FASTPFOR-D1. We omit S4-BP128-DM when it does not differ significantly from S4-BP128-D4.

scheme	bits/int	time	bits/int	time	bits/int	time
	GOV2 (sorted)		GOV2 (unsorted)		CLUEWEB09	
entropy	1.9		4.6		3.1	
NO BITMAP						
VARINT	8.2	6.3	8.5	8.2	8.2	17.5
FASTPFOR	3.8	6.3	6.2	8.0	5.0	17.3
S4-BP128-D4	6.2	2.1	7.9	2.5	7.5	5.6
S4-BP128-DM	6.2	2.1	7.7	2.6	7.4	5.7
S4-BP128-D2	5.6	2.2	7.4	2.6	7.1	6.0
S4-BP128-D1	5.0	2.4	6.9	2.8	6.6	6.5
S4-FASTPFOR	3.8	3.5	6.2	4.0	5.0	9.7
$B = 8$						
VARINT	5.8	3.0	7.4	4.7	6.2	9.9
FASTPFOR	4.3	3.0	6.3	4.5	5.2	9.3
S4-BP128-D4	5.7	1.2	7.4	1.6	6.5	3.4
S4-BP128-DM	5.7	1.2	7.3	1.6	6.4	3.4
S4-BP128-D2	5.5	1.3	7.1	1.6	6.3	3.6
S4-BP128-D1	5.3	1.4	6.8	1.7	6.1	3.8
S4-FASTPFOR	4.3	1.9	6.3	2.3	5.2	5.4
$B = 16$						
VARINT	6.3	2.1	8.2	3.2	6.9	6.7
FASTPFOR	5.5	2.1	7.6	2.9	6.4	5.9
S4-BP128-D4	6.5	0.9	8.4	1.2	7.2	2.5
S4-BP128-D2	6.4	1.0	8.1	1.2	7.1	2.6
S4-BP128-D1	6.2	1.1	7.9	1.3	7.0	2.7
S4-FASTPFOR	5.5	1.4	7.6	1.6	6.4	3.7
$B = 32$						
VARINT	8.2	1.4	11.1	2.0	8.9	4.2
FASTPFOR	7.8	1.5	10.9	1.8	8.7	3.7
S4-BP128-D4	8.4	0.8	11.2	0.9	9.1	1.9
S4-BP128-D2	8.4	0.8	11.1	1.0	9.1	2.0
S4-BP128-D1	8.3	0.8	11.0	1.0	9.0	2.0
S4-FASTPFOR	7.8	1.1	10.9	1.2	8.7	2.6
Skipper [35]	10.2	2.6	10.5	4.3	10.2	7.7

14. Barbay J, López-Ortiz A, Lu T, Salinger A. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics* Jan 2010; **14**:7:3.7–7:3.24, doi:10.1145/1498698.1564507.
15. Kane A, Tompa F. Skewed partial bitvectors for list intersection. *Proceedings of the 37th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2014.
16. Ding S, He J, Yan H, Suel T. Using graphics processors for high performance IR query processing. *Proceedings of the 18th international conference on World wide web*, WWW '09, ACM: New York, NY, USA, 2009; 421–430, doi:10.1145/1526709.1526766.
17. Ao N, Zhang F, Wu D, Stones DS, Wang G, Liu X, Liu J, Lin S. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proc. VLDB Endow.* May 2011; **4**(8):470–481.
18. Konow R, Navarro G, Clarke CL, López-Ortíz A. Faster and smaller inverted indices with treaps. *Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '13, ACM: New York, NY, USA, 2013; 193–202, doi:10.1145/2484028.2484088.
19. Vigna S. Quasi-succinct indices. *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, ACM: New York, NY, USA, 2013; 83–92.

20. Ottaviano G, Venturini R. Partitioned Elias-Fano indexes. *SIGIR '14*, ACM Press: New York, NY, USA, 2014.
21. Fog A. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Technical Report*, Copenhagen University College of Engineering 2014. http://www.agner.org/optimize/instruction_tables.pdf [last checked March 2015].
22. Intel Corporation. The Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/ [last checked March 2015] 2014.
23. Thiel L, Heaps H. Program design for retrospective searches on large data bases. *Information Storage and Retrieval* 1972; **8**(1):1–20.
24. Goldstein J, Ramakrishnan R, Shaft U. Compressing relations and indexes. *Proceedings of the Fourteenth International Conference on Data Engineering*, ICDE '98, IEEE Computer Society: Washington, DC, USA, 1998; 370–379.
25. Anh VN, Moffat A. Index compression using 64-bit words. *Softw. Pract. Exper.* 2010; **40**(2):131–147, doi: 10.1002/spe.v40:2.
26. Zukowski M, Heman S, Nes N, Boncz P. Super-scalar RAM-CPU cache compression. *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, IEEE Computer Society: Washington, DC, USA, 2006; 59–71, doi:10.1109/ICDE.2006.150.
27. Yan H, Ding S, Suel T. Inverted index compression and query processing with optimized document ordering. *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, ACM: New York, NY, USA, 2009; 401–410, doi:10.1145/1526709.1526764.
28. Bentley JL, Yao ACC. An almost optimal algorithm for unbounded searching. *Inf. Process. Lett.* 1976; **5**(3):82–87.
29. Schlegel B, Willhalm T, Lehner W. Fast sorted-set intersection using SIMD instructions. *Proceedings of the 2nd International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, ADMS '11, 2011.
30. Katsov I. Fast intersection of sorted lists using SSE instructions. http://highlyscalable.wordpress.com/2012/06/05/fast-intersection-sorted-lists-sse/ [last checked March 2015] 2009.
31. Balkesen C, Alonso G, Teubner J, Özsu M. Multicore, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.* 2013; **7**(1):85–96.
32. Schlegel B, Gemulla R, Lehner W. k-ary search on modern processors. *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, ACM: New York, NY, USA, 2009; 52–60, doi: 10.1145/1565694.1565705.
33. Polychroniou O, Ross KA. Vectorized bloom filters for advanced simd processors. *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, ACM: New York, NY, USA, 2014; 1–6, doi:10.1145/2619228.2619234.
34. Tonellotto N, Macdonald C, Ounis I. Effect of different docid orderings on dynamic pruning retrieval strategies. *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, ACM: New York, NY, USA, 2011; 1179–1180, doi:10.1145/2009916.2010108.
35. Sanders P, Transier F. Intersection in integer inverted indices. *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, ALENEX '07, SIAM, 2007; 71–83.
36. Willhalm T, Oukid I, Müller I, Faerber F. Vectorizing database column scans with complex predicates. *Proceedings of the 4th International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*, ADMS '13, 2013; 1–12.

A. SAMPLE C++ CODE USING INTEL INTRINSICS FOR INTERSECTION ALGORITHMS

```
uint32_t *r // pointer to short list
uint32_t *f // pointer to long list

// load T = 8 integers from f
__m128i F = _mm_loadu_si128((__m128i*)f);
__m128i G = _mm_loadu_si128((__m128i*)f+1);

// replicate current value from r
__m128i R = _mm_set1_epi32(*r);

// compare R and F, G
__m128i T0 =  _mm_cmpeq_epi32(F,R);
__m128i T1 =  _mm_cmpeq_epi32(G,R);
T = _mm_or_si128 (T0,T1);
/*
  if SSE4 is supported, we can instead use
  a slightly faster instruction _mm_testz_si128(T, T)
*/
```

Table VIII. Time required to answer queries from TREC 1MQ using bitmaps (HYB+M2) and short lists compressed with VARINT. We compare a skipping approach over short lists using blocks of 32 or 256 integers (as in Kane and Tompa [15]) vs. an approach using galloping over partitioned data.

	GOV2 sorted	GOV2 unsorted	CLUEWEB09
	$B = 8$		
galloping	3.0	4.7	9.9
skipping (32)	4.3	7.3	11
skipping (256)	4.4	7.5	12
	$B = 16$		
galloping	2.1	3.2	6.7
skipping (32)	2.5	4.2	6.4
skipping (256)	2.6	4.3	6.9
	$B = 32$		
galloping	1.4	2.0	4.2
skipping (32)	1.3	2.1	3.6
skipping (256)	1.4	2.1	3.8

```
if (_mm_movemask_epi8(T) != 0) {
  //one value in R matches a value in F, G
}
```

B. BITMAP-LIST HYBRIDS (HYB+M2) WITH SKIPPING

In § 7.7, we describe experiments using the HYB+M2 model. Short lists are first decompressed and then intersected (e.g., using the galloping algorithm). As recommended by Culpepper and Moffat [5], we do not use skipping [11]: all integers from the short lists are decompressed. In contrast, Kane and Tompa [15] found skipping over large blocks in the HYB+M2 context to be highly beneficial. For VARINT only, we adopt the Kane-Tompa approach, replacing galloping and partitions by skipping with blocks of 32 and 256 integers, and present the results in Table VIII using the same tests as in § 7.7. We see that skipping is worse than galloping in most cases (up to 60 %) though it can be sometimes be slightly beneficial (up to 15 %). Though these results are only for VARINT compression, we expect the benefits of skipping to be less for faster compression schemes such as S4-BP128-D4.