Efficient Lightweight Compression Alongside Fast Scans

Orestis Polychroniou
Columbia University
orestis@cs.columbia.edu

Kenneth A. Ross*
Columbia University
kar@cs.columbia.edu

ABSTRACT

The increasing main-memory capacity has allowed query execution to occur primarily in main memory. Database systems employ compression, not only to fit the data in main memory, but also to address the memory bandwidth bottleneck. Lightweight compression schemes focus on efficiency over compression rate and allow query operators to process the data in compressed form. For instance, dictionary compression keeps the distinct column values in a sorted dictionary and stores the values as index codes with the minimum number of bits. Packing the bits of each code contiguously, namely horizontal bit packing, has been optimized by using SIMD instructions for unpacking and by evaluating predicates in parallel per processor word for selection scans. Interleaving the bits of codes, namely vertical bit packing, provides faster scans, but incurs prohibitive costs for packing and unpacking. Here, we improve packing and unpacking for vertical bit packing using SIMD instructions, achieving more than an order of magnitude speedup. Also, we optimize horizontal bit packing on the latest CPUs and compare all approaches. While no single variant is better in all cases, vertical bit packing offers a good trade-off by combining the fastest scans with comparably fast packing and unpacking.

1. INTRODUCTION

The increase in main-memory capacity has allowed small to medium-sized databases to fit in main memory and the bottleneck has shifted from disk access to the memory bandwidth. In-memory query execution has raised the bar and vendors strive to provide real-time evaluation of analytical queries, even if a large portion of the database is accessed.

Column stores are a structural evolution of analytical database systems to adapt to the memory bandwidth bottleneck. By storing each column separately, we maximize the portion of useful data transferred for queries that access a few columns. To optimize the data fetching rate, we maximize both the useful data per transfer and the rate of transfers.

*Supported by NSF grant IIS-1422488 and an Oracle gift.

DaMoN '15, June 01, 2015, Melbourne, VIC, Australia.

http://dx.doi.org/10.1145/2771937.2771943.

Sequential memory scans are preferred over random accesses. Dropping the indexes altogether in favor of linear table scans is now a sensible design [21]. Cache-conscious joins, sorting, and aggregation also eliminate non-sequential accesses, while facilitating thread scalability. By tuning each operator to the underlying hardware, database systems maximize performance close to the hardware limit. In this context, memory bandwidth is the most important bottleneck.

Analytical databases are compressed, not only to fit the data in main memory, but also to improve beyond the memory bandwidth. Compression schemes allow operators to process the data directly on the compressed format [21, 24]. The most common scheme is dictionary compression. In its pure form, the n distinct values of each column are kept in a sorted dictionary and instead of storing the actual column values of each tuple, we store the dictionary indexes as codes using $\lceil \log n \rceil$ bits. Joins and selections can process the compressed codes instead of the actual values. Predicate constants in the query are also converted into codes that maintain the same order, by using the sorted dictionary.

Various techniques exist for packing the dictionary codes. In the standard horizontal bit packing, we can accelerate scans by wasting space to ensure word-alignment and allow for in-register data-parallel predicate evaluation [8]. Otherwise, we can still unpack the fully packed codes efficiently using SIMD instructions [24]. To achieve optimal scan performance by even skipping some parts of the data, we must pack the bits vertically [12] by interleaving them. However, the vertical layout is too expensive to generate and incurs prohibitive costs for extracting a large portion of tuples.

In this paper we study both the horizontal and the vertical bit packing layout, by focusing on three operations: packing, unpacking, and scans. We optimize the horizontal layout, including the word-aligned variant, using instructions that are only available in the latest mainstream CPUs and achieve significantly better instruction efficiency. Our novel contribution is the design and implementation of packing and unpacking for the vertical bit packing layout using SIMD instructions, which increases performance by more than an order of magnitude. Finally, we compare all approaches and highlight strengths and weaknesses. While horizontal bit packing is highly competitive, vertical bit packing combines the fastest scans with comparably fast packing and unpacking. Our work offers deep insights on the layouts and extends the trade-off options for lightweight database compression.

Section 2 presents related work. Sections 3 and 4 discuss horizontal and vertical bit packing respectively. Section 5 is our experimental evaluation and we conclude in Section 6.

2. RELATED WORK

Modern analytical databases are tuned for main-memory accesses [14]. Zukowski et al. discussed RAM to CPU-cache decompression [27]. Abadi et al. incorporated compression into query execution [1]. Holloway studied schema-tuned query code generation [5]. Johnson et al. studied selection scans on bit packed data [8]. Willhalm et al. optimized bit unpacking [24]. Raman et al. discussed entropy compression [20] and applied multiple techniques [21]. Mülhbauer et al. optimized data loading from CSV files [15]. Li et al. proposed vertical bit packing [12] and denormalization with compression to convert complex queries into scans [13]. Feng et al. studied aggregation functions on bit packed data [4].

Many compression schemes focus on efficiency. Anh et al. proposed pattern-based compression in processor words [2], Schlegel et al. studied skew-friendly compression [22], Stepanov studied vectorized variable-byte encoding [23], and Lemire et al. discussed vectorized frame-of-reference [11].

SIMD instructions have been used in many database operators. Zhou et al. used SIMD in linear scans, index scans, and nested-loop joins [26]. Inoue et al. used SIMD in combsort [6] and Chhugani et al. in mergesort [3]. Kim et al. studied multi-way trees [9]. Inoue et al. optimized sorted set intersections [7]. Polychroniou et al. studied aggregate updates [16], range indexes [17], Bloom filters [18], hash tables and partitioning used in radixsort and hash joins [19].

3. HORIZONTAL BIT PACKING

Horizontal bit packing stores an array of integer codes using a constant number of bits. We need $b = \lceil log n \rceil$ bits for codes in $[0, n)$. The bits per code are stored contiguously.

3.1 Scalar Packing & Unpacking

Packing 32-bit codes to b-bit codes is shown below in scalar C code. We iterate over the unpacked codes and construct the next packed word using a 64-bit scalar register.

```
void pack(const uint32_t *unpacked, int8_t bits,
          size_t codes, uint64_t *packed) {
 size_t i, o; uint64_t word = 0; int8_t word_bits = 0;
 for (o = i = 0;  i != codes;  i++) {
   uint64_t code = unpacked[i];
   word |= code << word_bits;   word_bits += bits;
   if (word_bits >= 64) {
     packed[o++] = word;   word_bits -= 64;
     word = code >> (bits - word_bits); }}
```

Unpacking in scalar C code is shown below. We iterate over the codes and branch to fetch the next packed word. Both cases are simple and are shown for completeness;

```
void unpack(const uint64_t *packed, int8_t bits,
            size_t codes, uint32_t *unpacked) {
 uint64_t word = 0, mask = (1 << bits) - 1;
 size_t i, o;  int8_t word_bits = 0;
 for (i = o = 0;  o != codes;  o++) {
   uint64_t code = word & mask;
   word >>= word_bits;   word_bits -= bits;
   if (word_bits < 0) {
     uint64_t word_2 = *packed[i++];
     word = word_2 >> -word_bits;   word_bits += 64;
     code |= (word_2 << word_bits) >> (64 - bits); }
   unpacked[o] = code; }}
```

We can optimize the above methods for each b individually, by using constant stride shifts and by removing the branch [11, 27]. We do not evaluate such optimizations here.

3.2 Word-Aligned Scalar Scanning

Based on previous work [8, 10], scanning the packed format to evaluate selective predicates can be done directly on the packed format without unpacking. The $C_1 \neq C_2$ boolean expression in b-bit arithmetic is equal to the overflow bit of $(C_1 \texttt{ xor } C_2) + (2^b - 1)$. The $C_1 < C_2$ expression is equal to the overflow bit of $(C_1 \texttt{ xor } (2^b - 1)) + C_2$. To parallelize the process, we pack $b' = \lfloor w/(b+1) \rfloor$ codes per processor word using $b+1$ bits per code. The extra bit of each code is used as the overflow bit and allows predicate evaluation to occur in parallel for all the b' codes packed in the processor word, by ensuring that the addition does not propagate a carry to the $b+1$ bits of the next packed code. Packing and unpacking become simpler by using an inner loop for the b' codes. However, the remaining $w - b' \cdot (b+1)$ bits at the end of each word are ignored, on top of the extra bit per code.

```
void scan(const uint64_t *packed, int8_t bits,
          size_t codes, uint64_t *bitmap,
          uint32_t min, uint32_t max) {
 /* repeat constants using given bit range */
 uint64_t mask_min  = repeat(bits + 1, min)
 uint64_t mask_max  = repeat(bits + 1, max);
 uint64_t mask_1_0n = repeat(bits + 1, 1ull << bits);
 uint64_t mask_0_1n = mask_1_0n - (mask_1_0n >> bits);
 /* full horizontal bit packing for the bitmap */
 int8_t bm_wb = 0, bm_b = 64 / (bits + 1);
 uint64_t bm_w = 0;  size_t i, o, words = codes / bm_b;
 for (o = i = 0;  i != words;  i++) {
   uint64_t word = packed[i++];
   /* evaluate column < min and max < column */
   uint64_t lt_min = mask_min + (word ^ mask_0_1n);
   uint64_t gt_max = word + (mask_max ^ mask_0_1n);
   /* combine, extract, and compact resulting bits */
   word = _pext_u64(lt_min | gt_max, mask_1_0n);
   /* pack the predicate result bits into the bitmap */
   bm_w |= word << bm_wb;   bm_wb += bm_b;
   if (bm_wb >= 64) {
     bitmap[o++] = ~bm_w;   bm_wb -= 64;
     bm_w = word >> (bm_bits - bm_wb); }}
```

Scalar C code for scanning the word-aligned layout using the $C_1 \leq column \leq C_2$ predicate is shown above. The helper function `repeat(b,x)` replicates `x` every `b` bits. The overflow bits are extracted and compacted in the low bits of the output register using the `pext` instruction (`_pext_u64`) supported only by the latest CPUs. With so few instructions per processor word, scanning performance is reduced proportionally to the wasted space. Using this technique on the fully packed format is possible, but is not evaluated here.

3.3 SIMD Unpacking & Scanning

Based on previous work [24], unpacking the codes can be done fast, even if no space is wasted. By using register shuffling, we can replicate the packed bytes and unpack one code per SIMD lane [24]. If each SIMD register fits l codes, we unpack l codes at a time using the following steps:

1. Load the next $l \cdot b$ bits from the input in a vector. If $l = 8$, load the next b bytes using a byte-aligned vector load.
2. Shuffle the vector bytes to align codes at byte boundaries. Bytes $4 \cdot i$ to $4 \cdot i + 3$ of the i-th 32-bit lane are pulled from bytes j to $j + 3$ of the loaded vector, where $j = \lfloor i \cdot b/8 \rfloor$.
3. Shift the 32-bit lanes to align codes at bit boundaries. The i-th 32-bit lane is shifted right by $i \cdot b$ modulo 8.
4. Bitwise-`and` the 32-bit lanes with $2^b - 1$ to eliminate the high bits that belong to the next codes.
5. Use or store the l unpacked 32-bit lanes.

The SIMD code for unpacking horizontally bit packed codes using 256-bit SIMD (AVX 2) is shown below. In our evaluation, we manually unroll the loop four times to eliminate most instruction latencies and maximize IPC. The functions with the _mm prefix are intrinsics for SIMD instructions. We omit describing each instruction here due to limited space. Detailed descriptions are available online.[1]

Specifically for the SIMD ISA that we use (AVX 2), byte shuffling gathers bytes from each half of the register in isolation. To shuffle bytes across all lanes, we copy each half to both, shuffle bytes twice, and blend. If $b \leq 16$, we use one shuffle instruction and a 128-bit load instead of a 256-bit load. If $b \in \{27, 29, 30, 31\}$, we use variable stride shifts in 64-bit lanes, because packed codes can span across 5 bytes.

```
void unpack(const uint64_t *packed, int8_t bits,
            size_t codes, uint32_t *unpacked) {
 int8_t shuffle[32]; uint32_t shift[8]; size_t i, j, o;
 /* generate byte shuffle and bit shifting masks */
 for (i = 0;  i != 8;  i++) {
   for (j = 0;  j != 4;  j++) {
     shuffle[i * 4 + j] = (i * bits) / 8 + j; }
   shift[i] = (i * bits) % 8; }
 __m256i mask_bits_lo = _mm256_loadu_si256(shift);
 __m256i mask_bits_hi = _mm256_set1_epi32((1<<bits)-1);
 __m256i mask_bytes_lo = _mm256_loadu_si256(shuffle);
 __m256i mask_bytes_hi = _mm256_sub_epi8(mask_bytes_lo,
                         _mm256_set1_epi8(16));
 /* byte alignment for input (unaligned SIMD loads) */
 const int8_t *packed_b = (const int8_t*) packed;
 if (bits <= 16) { /* 4+2 instr. (for load & store) */
   for (i = o = 0;  o != codes;  i += bits, o += 8) {
     /* load 128 bits and broadcast low part (cast) */
     __m256i data = _mm_loadu_si128(&packed_b[i]));
     data = _mm256_permute4x64_epi64(data, 0x44);
     /* shuffle bytes to align by 32-bit lanes */
     data = _mm256_shuffle_epi8(data, mask_bytes_lo);
     /* align ints and clear high order bits */
     data = _mm256_srlv_epi32(data, mask_bits_lo);
     data = _mm256_and_si256(data, mask_bits_hi);
     _mm256_store_si256(&unpacked[o], data); }}
 else if (bits <= 26 || bits == 28) { /* 7+2 instr. */
   for (i = o = 0;  o != codes;  i += bits, o += 8) {
     /* load 256 bits and broadcast both parts */
     __m256i data = _mm256_loadu_si256(&packed_b[i]);
     __m256i lo = _mm256_permute4x64_epi64(data, 0x44);
     __m256i hi = _mm256_permute4x64_epi64(data, 0xEE);
     /* shuffle low and high 128 bits separately  */
     lo = _mm256_shuffle_epi8(lo, mask_bytes_lo);
     hi = _mm256_shuffle_epi8(hi, mask_bytes_hi);
     data = _mm256_blendv_epi8(lo, hi, mask_bytes_hi);
     /* align ints and clear high order bits */
     data = _mm256_srlv_epi32(data, mask_bits_lo);
     data = _mm256_and_si256(data, mask_bits_hi);
     _mm256_store_si256(&unpacked[o], data); }}
 else { [...] /* 12+2 instr. (64-bit shift masks) */ }}
```

With 256-bit SIMD (AVX 2), we need 4 SIMD instructions per 8 codes if $b \leq 16$, 7 instructions if $b \leq 26$ or $b = 28$, and 12 instructions if $b \in \{27, 29, 30, 31\}$. Evaluating the $C_1 \leq column \leq C_2$ predicate and extracting a bitmask adds 4 more instructions. Previous work using 128-bit SIMD (SSE) reported 10 instructions per code [12], partially due to emulating variable stride shifts via multiplications.

By viewing the input as l interleaved streams, we can extend the optimized per b packing and unpacking [27] to use SIMD [11]. Each stream is processed by one of the l SIMD lanes. Packing gets faster if not memory bound, but unpacking saves very few instructions compared to the above.

[1] http://software.intel.com/sites/landingpage/IntrinsicsGuide/

4. VERTICAL BIT PACKING

In vertical bit packing [12], the b bits per code are not stored contiguously, but are interleaved for groups of k codes. The bits of the next k codes are interleaved in b k-bit words and the i-th word has the i-th bit of each code. The layout allows predicate evaluation on the packed format and executes at most the same order of instructions as word-aligned horizontal bit packing (Section 3.2), but without wasting any bits. Word alignment is guaranteed by setting k so that $w|k$.

4.1 Scalar Packing & Unpacking

Scalar C code for vertical bit packing with $k = 64$ is shown below. We extract one bit at a time and add it to the packed word. The process is repeated b times per code, thus executing $O(nb)$ operations. The high order bits are stored first.

```
void pack(const uint32_t *unpacked, int8_t bits,
          size_t codes, uint64_t *packed) {
 size_t i, o;  int8_t b1, b2;  uint64_t word;
 for (o = i = 0;  i != codes;  o += bits, i += 64) {
   for (b1 = bits - 1;  b1 >= 0;  b1--) {
     for (b2 = 63;  b2 >= 0;  b2--) {
       uint64_t code = unpacked[i + b2];
       word += word + ((code >> b1) & 1); }
     packed[o + b1] = word; }}}
```

Unpacking the vertical layout is symmetrical to packing. Scalar C code using the same parameters is shown below. We build each unpacked code by extracting one bit per packed word at a time, again executing $O(nb)$ operations.

```
void unpack(const uint64_t *packed, int8_t bits,
            size_t codes, uint32_t *unpacked) {
 size_t i, o;   int8_t b1, b2;
 for (i = o = 0;  o != codes;  i += bits, o += 64) {
   for (b1 = 0;  b1 != 64;  b1++) {
     uint64_t code = 0;
     for (b2 = 0;  b2 != bits;  b2++) {
       uint64_t word = packed[i + b2];
       code += code + ((word >> b1) & 1); }
     unpacked[o + b1] = code; }}}
```

The code we measure is further optimized, but the vertical layout can still be more than an order of magnitude slower than the horizontal, for both packing and unpacking.

4.2 SIMD Packing

The simplest version of SIMD vertical bit packing holds all k unpacked codes in SIMD registers and extract one bit per lane from each SIMD register. The process is repeated b times for b-bit codes. If the CPU core provides 16 256-bit registers, we can use 8 to hold $k = 64$ unpacked codes. We process k unpacked codes at a time (l is the number of 32-bit SIMD register lanes) and repeat the following steps:

1. Load the next k words in k/l SIMD registers.
2. Shift left the loaded k/l SIMD registers by $32 - b$.
3. Repeat b times:
   (a) Extract the sign bits per 32-bit lane from the k/l SIMD registers into k/l scalar l-bit bitmasks.
   (b) Append k/l bitmasks (k bits) to the output.
   (c) Double the k/l SIMD registers.

SIMD code is significantly faster than scalar code, even though the complexity remains $O(nb)$. First, avoid reloading the unpacked codes several times from the L1 cache but keep them resident in SIMD registers instead. Second, we extract multiple bits per instruction, executing b/l bit extraction instructions per code rather than b. For 256-bit SIMD, $l = 8$.

Compared to the scalar code, we execute more than an order of magnitude less instructions. To further improve the number of bits extracted per instruction, we pack each byte of the packed codes in fewer SIMD registers and extract bits using 8-bit lanes. The steps per k codes are shown below:

1. Load the next k words in k/l SIMD registers.
2. Shift left the loaded k/l SIMD registers by $32-b$ bits.
3. Repeat $\lceil b/4 \rceil$ times:
   (a) Pack the k/l registers of k 32-bit codes into $k/4l$ registers of k 8-bit codes using the most significant byte.
   (b) Repeat 8 times or until out of bits:
      i. Extract the sign bits per byte lane from the $k/4l$ SIMD registers into $k/4l$ scalar $4l$-bit bitmasks.
      ii. Append the bitmasks (k bits) to the output.
      iii. Double the $k/4l$ SIMD registers.
   (c) Shift the k/l registers right by 8 bits per 32-bit lane.

```
void pack(const uint32_t *unpacked, int8_t bits,
          size_t codes, uint64_t *packed) {
 __m128i shift = _mm_cvtsi32_si128(32 - bits);
 __m256i order = _mm256_set_epi32(7,3,6,2,5,1,4,0);
 size_t i, o;   int8_t b1, b2;
 for (o = i = 0;  i != codes;  i += 64) {
   /* load 64 ints in 8 256-bit SIMD registers */
   __m256i r1 = _mm256_load_si256(&unpacked[i]);
   [...] /* 7 symmetrical lines omitted */
   /* shift left to move b-th bit as MSB (sign bit) */
   r1 = _mm256_sll_epi32(r1, shift);
   [...] /* 7 symmetrical lines omitted */
   for (b1 = 0;  b1 + 8 < bits;  b1 += 8) {
     /* isolate the high order byte as the low byte */
     __m256i s1 = _mm256_srli_epi32(r1, 24);
     [...] /* 7 symmetrical lines omitted */
     /* pack 32-bit into 16-bit in 1/2 of registers */
     __m256i s12 = _mm256_packus_epi32(s1, s2);
     [...] /* 3 symmetrical lines omitted */
     /* pack 16-bit into 8-bit in 1/4 of registers */
     __m256i s1234 = _mm256_packus_epi16(s12, s34);
     __m256i s5678 = _mm256_packus_epi16(s56, s78);
     /* restore order polluted by pack instructions */
     s1234 = _mm256_permutevar8x32_epi32(s1234, order);
     s5678 = _mm256_permutevar8x32_epi32(s5678, order);
     for (b2 = min(bits - b1, 8);  b2 != 0;  b2--) {
       /* extract one bit at a time per byte lane */
       uint32_t lo = _mm256_movemask_epi8(s1234);
       uint32_t hi = _mm256_movemask_epi8(s5678);
       s1234 = _mm256_add_epi8(s1234, s1234);
       s5678 = _mm256_add_epi8(s5678, s5678);
       packed[o++] = append_2x32(lo, hi); }
     /* shift the input registers left by 1 byte */
     r1 = _mm256_slli_epi32(r1, 8);
     [...] /* 7 symmetrical lines omitted */ }}
```

C with 256-bit SIMD (AVX2) is shown ($k = 64$). Using 8-bit instead of 32-bit lanes to extract bits, reduces the SIMD instructions per code from $(16b + 8)/64$ to $(10b + 30)/64$.

4.3 SIMD Unpacking

Unpacking the vertical bit packed format is based on the same principles as packing, but performs the inverse operation. Instead of extracting one bit per SIMD lane, we convert the packed bits to integer masks and insert one bit per SIMD lane in the registers that hold the unpacked words.

1. Set the next k codes in k/l SIMD registers to zero.
2. Repeat b times:
   (a) Load the next k bits from the input.
   (b) Convert k bits to {0,-1} 32-bit masks in k/l registers.
   (c) Double the k/l registers and subtract the 32-bit masks.
3. Append the k/l registers to the output.

Similarly to bit packing, we do not reload the input multiple times but keep the output unpacked codes in SIMD registers. To improve the number of bits inserted per instruction, we use smaller SIMD lanes again. We extract bits in byte lanes that are then up-converted and inserted in the k/l output registers. The steps per k codes are shown below:

1. Set the next k codes in k/l SIMD registers to zero.
2. Repeat $\lceil b/4 \rceil$ times:
   (a) Reset $k/4l$ SIMD registers to zero.
   (b) Repeat 8 times or until out of bits:
      i. Load the next k bits from the input.
      ii. Convert k bits to {0,-1} 8-bit masks in $k/4l$ registers.
      iii. Double the $k/4l$ registers and subtract the 8-bit masks.
   (c) Shift the k/l registers left by 8 bits per 32-bit lane.
   (d) Up-convert 8-bit lanes in $k/4l$ registers to 32-bit lanes.
   (e) Bitwise-or the up-converted registers with the results.
3. Append the k/l registers to the output.

```
void unpack(const uint64_t *packed, int8_t bits,
            size_t codes, uint32_t *unpacked) {
  size_t i, o;   int8_t b;   uint64_t
  /* masks used to convert bits into byte masks */
    m4=0x0404040404040404ull, m8=0x0808080808080808ull,
    mC=0x0C0C0C0C0C0C0C0Cull, mP=0x8040201008040201ull;
  __m256i shuf_lo = _mm256_set_epi64x(m8, 0,m8, 0);
  __m256i shuf_hi = _mm256_set_epi64x(mC,m4,mC,m4);
  __m256i bif_offset = _mm256_set1_epi64x(mP);
  for (i = o = 0;  o != codes;  o += 64) {
    /* initialize unpacked words by setting to 0 */
    __m256i r1 = _mm256_setzero_si256();
    [...] /* 7 symmetrical lines omitted */
    for (b = bits;  b != 0;  ) {
      __m256i r_lo = _mm256_setzero_si256();
      __m256i r_hi = _mm256_setzero_si256();
      for (;  (b & 7) != 0;  b--) {
        /* load 64 bits and replicate to 64 bytes */
        __m128i b_cmp = _mm_loadl_epi64(&packed[i++]);
        __m256i b = _mm256_cvtepu8_epi32(b_cmp);
        __m256i b_lo = _mm256_shuffle_epi8(b, shuf_lo);
        __m256i b_hi = _mm256_shuffle_epi8(b, shuf_hi);
        /* convert target bit to {0,-1} mask per byte */
        b_lo = _mm256_and_si256(b_lo, bit_offset);
        b_hi = _mm256_and_si256(b_hi, bit_offset);
        b_lo = _mm256_cmpeq_epi8(b_lo, bit_offset);
        b_hi = _mm256_cmpeq_epi8(b_hi, bit_offset);
        /* shift left and add new bit per byte lane */
        r_lo = _mm256_add_epi8(r_lo, r_lo);
        r_hi = _mm256_add_epi8(r_hi, r_hi);
        r_lo = _mm256_sub_epi8(r_lo, b_lo);
        r_hi = _mm256_sub_epi8(r_hi, b_hi); }
      /* shift the output registers left by 1 byte */
      r1 = _mm256_slli_epi32(r1, 8);
      [...] /* 7 symmetrical lines omitted */
      /* up-convert first 8 bytes to ints (cast) */
      __m256i b_lo = _mm256_cvtepu8_epi32(r_lo);
      __m256i b_hi = _mm256_cvtepu8_epi32(r_hi);
      /* merge 1 byte per code with the result */
      r1 = _mm256_or_si256(r1, b_lo);
      r2 = _mm256_or_si256(r2, b_hi);
      /* rotate to process next 8 bytes */
      r_lo = _mm256_permute4x64_epi64(r_lo, 0x39);
      r_hi = _mm256_permute4x64_epi64(r_hi, 0x39);
      [...] /* 18 symmetrical lines omitted */ }
    /* store unpacked codes to output */
    _mm256_store_si256(&unpacked[o], r1);
    [...] /* 7 symmetrical lines omitted */ }}
```

C code with 256-bit SIMD (AVX) intrinsics for unpacking is shown below. We execute $18b + 30$ SIMD instructions per 64 codes, which is $O(nb)$. However, in practice we get comparable performance to $O(n)$ SIMD horizontal unpacking.

4.4 Scanning & Bandwidth Saving

Predicate evaluation on vertically bit packed data can proceed directly on the packed format without unpacking. Equality to a constant is evaluated as the bitwise-`and` of the bitwise-`xnor` of all b bits of the k words with the bits of the constant. To evaluate "<" or ">", we find the highest order bit that the column value differs from the constant and copy the target bit of the constant, for "<", or its inverse, for ">".

If the processor register has w bits and $w|k$, we execute $O(nb/w)$ operations during scanning. However, we do not need to access all b w-bit words used to pack w codes. Instead, once a bit differs for *all* w codes, we can stop earlier.

For the predicate: $column = C$, the probability for one code to reach the i-th word is: 2^{1-i} and for *any* of the w codes: $P(X \geq i) = 1 - (1 - 2^{1-i})^w$. For the predicate: $C_1 \leq column \leq C_2$, the probability for one code to reach the i-th word is: $1 - (1 - 2^{1-i})^2$ and for *any* of the w codes: $P(X \geq i) = 1 - (1 - [1 - (1 - 2^{1-i})^2])^w = 1 - (1 - 2^{1-i})^{2w}$. The expected number of w-bit words accessed for w codes using b bits is: $E[X] = \sum_{i=1}^{b} P(X = i) \cdot i = \sum_{i=1}^{b} P(X \geq i)$.

```
void scan(const uint64_t *packed, int8_t bits,
          size_t codes, uint64_t *bitmap,
          uint32_t min, uint32_t max) {
 /* shift constants to place target bit as sign bit */
 int64_t min_up = ((int64_t) min) << (64 - bits);
 int64_t max_up = ((int64_t) max) << (64 - bits);
 size_t i, o, bitmasks = codes / 64;  int8_t b;
 for (i = o = 0;  o != bitmasks;  i += bits, o++) {
   /* initialize bitmasks used as comparison results */
   int64_t gt_min = 0, eq_min = -1, const_min = min_up;
   int64_t lt_max = 0, eq_max = -1, const_max = max_up;
   for (b = 0;  b != bits;  b++) {
     int64_t word = packed[i + b];
     /* broadcast sign bit and check if different */
     int64_t bit_min = word ^ (const_min >> 63);
     int64_t bit_max = word ^ (const_max >> 63);
     /* update greater-than and less-than bits */
     gt_min |= eq_min & bit_min &  word;
     lt_max |= eq_max & bit_max & ~word;
     /* update equality and check for early exit */
     eq_min &= ~bit_min;
     eq_max &= ~bit_max;
     if ((eq_min | eq_max) == 0) break;
     const_min += const_min;  const_max += const_max; }
   /* store the comparison result in the bitmap */
   bitmap[o++] = (gt_min|eq_min) & (lt_max|eq_max); }}
```

Scalar code for scanning to evaluate $C_1 \leq column \leq C_2$ is shown above. The shifts by 63 broadcast the sign bit. Extending the code to use SIMD instructions is trivial if we use a larger k. In such a case, the number of instructions per code is reduced, but the register width w is increased and so is the number of w-bit words accessed per w codes.

Scanning less than b words does not improve performance, due to cache line granularity of RAM accesses and hardware prefetching. Bitweaving [12] addresses this problem by vertically packing $b' < b$ bits at a time separately. The extreme is to store each bit separately [25]. Here, we increase k to place low and high order bits of codes in distant cache lines.

Setting $k > w$ can be handled by all vertical bit packing operations with little overhead, if $w|k$. For packing, we store the w-bit packed words with a k/w stride. For unpacking and scanning, we load the packed w-bit words with a k/w stride. Processing the first w codes out of k would place packed words for the few next runs of w codes in the L1 cache and would be loaded from the L1 in the next runs. When k exceeds one cache line of L bits, in order to find the number of cache line accesses, we use the same $E[X]$ formula by setting $w = L$. Also, to compensate for hardware prefetching of neighbor cache lines, we increase k beyond one cache line up to the available L1 capacity. Finally, we can use SIMD for scanning without increasing the false prefetches.

When fewer tuples are extracted, performance differs when selectivity is low enough to skip cache lines. Then, vertical bit packing with $k \geq L$ can be $O(b)$ times slower than horizontal, since we access 1 cache line per code bit rather than 1–2 cache lines per code. We do not evaluate this case here.

5. EXPERIMENTAL EVALUATION

The platform we use has one Intel Xeon E3-1275v3 CPU at 3.5 GHz based on the Haswell micro-architecture. The CPU has 4 cores with 2-way SMT and supports 256-bit SIMD (AVX2). The RAM is dual-channel ECC DDR3 at 1600 MHz with 21.8 GB/s load bandwidth. We compile with ICC 15 using `-O3`. We use synthetic uniform random data, even though most methods with constant b are invariant to the input value distribution. Simple integer compression schemes, such as frame-of-reference, that vary b across small groups of codes, are trivial extensions adding little overhead.

Figure 1 measures packing, unpacking, and scanning for horizontal and vertical bit packing, by varying b and by using all hardware threads. The input is 10^9 codes. Output materialization to memory saturates the bandwidth for both packing and unpacking, thus is excluded and a checksum is computed instead. For scanning, we use $C_1 \leq column \leq C_2$ as selective predicate and include bitmap materialization.

Packing typically saturates the bandwidth by loading the unpacked input. Word-aligned horizontal packing is faster than the fully packed version. Packing to the vertical bit packed layout in SIMD code executes $O(nb)$ operations, but still saturates the bandwidth, even if $b = 32$. The vectorization speedup is 1.1–27.4X and is maximized when $b = 32$.

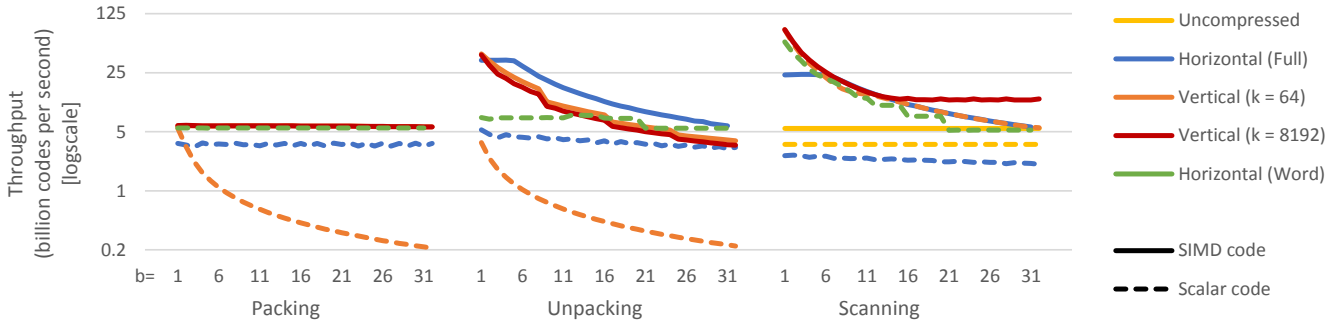

Figure 1: Multi-threaded packing, unpacking, and scanning for horizontal and vertical bit packing

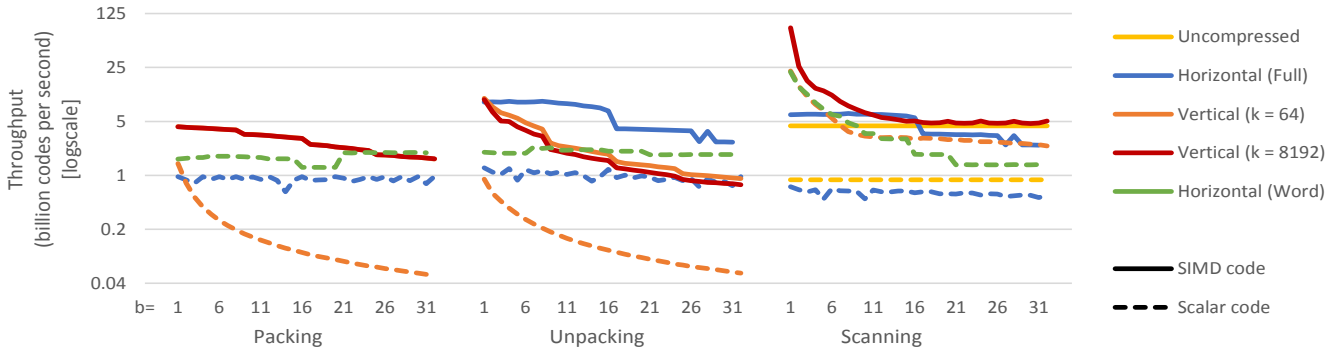

Figure 2: Single-threaded packing, unpacking, and scanning for horizontal and vertical bit packing

Unpacking the horizontal layout is faster than unpacking the vertical layout ($k = 64$) by 1.4–1.6X, if both are written in SIMD code, except for $b = 9$ and $b = 10$ that get 1.8X and 1.7X. The vectorization speedup is 1.9–8.4X for horizontal and 11–20X for the vertical layout. Increasing k to 8192 makes unpacking 10–15% slower. For larger b, vertical unpacking in scalar code is prohibitively slow. For smaller b, scalar unpacking of one code at a time is compute-bound.

Scanning the vertical layout in scalar code is faster than scanning the horizontal in SIMD by 1.1–3.4X for $b \leq 5$, while the opposite is true by 1.05–1.25X for $6 \leq b \leq 12$. For larger b, both methods saturate the bandwidth. The word-aligned layout loses performance by accessing a larger data footprint. Evaluating the predicates after horizontally unpacking in SIMD reduces performance by $\approx 30\%$ for $b \leq 5$ down to $\approx 5\%$ for $b > 16$. With $k = 64$, even if we access 8–9 words in 1–2 cache lines per 64 codes, we prefetch all cache lines. By increasing k beyond a cache line, we access 10–11 cache lines due to wider registers, but minimize the false prefetches. Performance is constant for larger b and outperforms all methods. For $b = 32$, scanning the vertical layout effectively doubles the bandwidth. Also, with $b = 32$, we can also encode the actual (32-bit) column values, not the dictionary codes, while still providing twice faster scans.

Figure 2 repeats the experiment of Figure 1 in a single thread to simulate higher memory bandwidth, making most methods compute-bound. Using multiple threads makes us memory-bound and reduces the gap between horizontal and vertical unpacking. Vertical scanning gets 1.9–3.6X speedup from SIMD and exposes the early exit for $b \geq 9$ if $k = 64$ and for $b \geq 12$ for $k = 8192$, matching the analysis of Section 4.4. Scanning uncompressed data gets a 5X speedup from SIMD and is close to the fastest, highlighting that lightweight compression is of limited use when we are not memory-bound.

6. CONCLUSION

We studied multiple bit packing layouts employed for lightweight compression. For horizontal bit packing, we considered both the word-aligned and the fully packed variant, and optimized previous techniques using the latest scalar and SIMD instructions. For vertical bit packing, we designed and implemented efficient packing and unpacking, by placing multiple codes in SIMD registers and by maximizing the number of bit movements per instruction. All variants are compared for packing, unpacking, and scanning. While the horizontal schemes are highly competitive, vertical bit packing combines the fastest scans with comparably fast packing and unpacking, extending the available trade-off options.

7. REFERENCES

[1] D. Abadi et al. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.
[2] V. N. Anh et al. Index compression using 64-bit words. *Software: Practice and Experience*, 40(2):131–147, Feb. 2010.
[3] J. Chhugani et al. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *VLDB*, pages 1313–1324, 2008.
[4] Z. Feng et al. Accelerating aggregation using intra-cycle parallelism. In *ICDE*, 2015.
[5] A. L. Holloway et al. How to barter bits for chronons: Compression and bandwidth trade offs for database scans. In *SIGMOD*, pages 389–400, 2007.
[6] H. Inoue et al. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, pages 189–198, 2007.
[7] H. Inoue et al. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, Nov. 2014.
[8] R. Johnson et al. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, Aug. 2008.
[9] C. Kim et al. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.
[10] L. Lamport. Multiple byte processing with full-word instructions. *CACM*, 18(8):471–475, Aug. 1975.
[11] D. Lemire et al. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, Jan. 2015.
[12] Y. Li et al. Bitweaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.
[13] Y. Li et al. WideTable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, June 2014.
[14] S. Manegold et al. Optimizing database architecture for the new bottleneck: Memory access. *J. VLDB*, 9(3):231–246, Dec. 2000.
[15] T. Mühlbauer et al. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, Sept. 2013.
[16] O. Polychroniou et al. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*, 2013.
[17] O. Polychroniou et al. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014.
[18] O. Polychroniou et al. Vectorized Bloom filters for advanced SIMD processors. In *DaMoN*, 2014.
[19] O. Polychroniou et al. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, 2015.
[20] V. Raman et al. How to wring a table dry: Entropy compression of relations and querying of compressed relations. In *VLDB*, pages 858–869, 2006.
[21] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, Aug. 2013.
[22] B. Schlegel et al. Fast integer compression using SIMD instructions. In *DaMoN*, pages 34–40, 2010.
[23] A. A. Stepanov et al. SIMD-based decoding of posting lists. In *CIKM*, pages 317–326, 2011.
[24] T. Willhalm et al. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, Aug. 2009.
[25] H. K. T. Wong et al. Bit transposed files. In *VLDB*, pages 448–457, 1985.
[26] J. Zhou et al. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.
[27] M. Zukowski et al. Super-scalar RAM-CPU cache compression. In *ICDE*, 2006.