Vectorizing Database Column Scans with Complex Predicates

Thomas Willhalm[‡]
thomas.willhalm@intel.com

Ismail Oukid[†‡]
i.oukid@sap.com

Ingo Müller[†*]
ingo.mueller@kit.edu

Franz Faerber[†]
franz.faerber@sap.com

[‡]Intel GmbH

[†]SAP AG

[*]Karlsruhe Institute of Technology

ABSTRACT

The performance of the full table scan is critical for the overall performance of column-store database systems such as the SAP HANA database. Compressing the underlying column data format is both an advantage and a challenge, because it reduces the data volume involved in a scan on one hand and introduces the need for decompression during the scan on the other hand. In previous work [26] we have shown how to accelerate the column-scan with range predicates using SIMD instructions. In this paper, we present a framework for vectorized scans with more complex predicates. One important building block is the In-List predicate, where all rows whose values are contained in a given list of values are selected. While this seems to exhibit only little data parallelism on first sight, we show that a performant vectorized implementation is possible using the new Intel AVX2 instruction set. We also improve our previous algorithms by leveraging the increased vector-width. Finally in a detailed performance evaluation, we show the benefit of these optimizations and of the new instruction set: in almost all cases our scans needs less than one CPU cycle per row including scans with In-List predicate, leading to an overall throughput of 8 billion rows per second and more on a single core.

1. INTRODUCTION

Today's processors implement a rich set of techniques to accelerate the performance of applications. Most prominent is the use of multiple cores per processor, where each core can execute an independent thread. Additionally, several vendors implement simultaneous multi-threading, where a single core executes two or more threads concurrently and thereby optimizes the usage of the execution units in the core [1, 8, 13]. But even within a single thread, several instructions can be executed inside a single clock cycle.

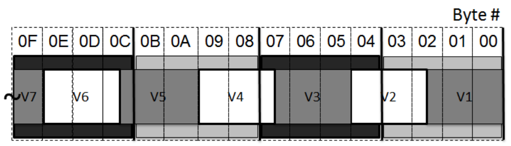

Figure 1: The codewords of a column are not aligned to machine word boundaries. They may be spread across several machine words and share their machine word(s) with other codewords.

The focus of this paper lies on SIMD, where a Single Instruction can perform on Multiple Data elements. SIMD instructions also benefit from multiple execution ports and out-of-order execution and can take advantage of multiple cores by parallel execution. For the evaluation of our algorithms, we use the Intel Advanced Vector Extensions 2 (Intel AVX2), the implementation of SIMD instructions in the current 4th generation Intel Core architecture. They feature a vector length of 256 bits and implement *vector-vector shift* and *gather* instructions, which play a fundamental role for the performance of our routines.

These SIMD instructions work on vectors of machine words such as 8, 16, 32, or 64bit integers. However if a column is compressed, its values are not materialized in this format in RAM. Instead, they may be stored in the commonly used combination of domain encoding and bit-compression: Each of the n distinct values of a column gets assigned an integer code between 0 and $n-1$ and each value of the column is replaced by its codeword. The mapping between real values and codewords is stored in a different datastructure. The sequence of codewords is stored in a bit stream where every codeword just uses a fixed number of $b = \lceil \log n \rceil$ bits. Figure 1 shows an example of a column encoded in such a bit stream. As the codewords do not start at machine word boundaries, some unpacking logic is necessary before the evaluation of the scan predicate.

The most common scans search values using an equality predicate, i.e., $=, \neq, >, \geq, <, \leq$. All of these can be reduced to a range predicate of the form $[min, max)$, which is the predicate we concentrated on in our previous work [26]. For the range scan, we had two variants: If the selectivity is very low, the result is materialized in form of a (short) list of row indices for the matches. In all other cases, the result is materialized as a bit vector where the 1s represent the matching rows. However, sometimes more sophisticated predicates are required. The scan predicate may contain an arithmetic expression, involve several columns, or constitute of a list of desired values like in the `IN` operator in SQL.

The main contribution of this work is a framework for evaluating almost arbitrarily complex predicates while preserving the highest possible performance. For that, we categorize the scan predicates into the following types:

Range predicates: All inequality predicates involving a single column can be expressed as a range predicate. The same is true for predicates involving arithmetic expressions on a single column where the expression can be inverted and applied to the range.

Vectorizable predicates: Many other predicates, including many arithmetic expressions on a single column, can easily be expressed using vector instructions. However, for good performance, it is important to tightly integrate unpacking and predicate evaluation. In our framework we have a template scan that can be extended by arbitrary vectorized predicates.

In-List predicates: If a list of desired values is given (from an `IN` operator or a subquery) or can be created (by applying the arithmetic expression to all values of a small dictionary), we translate the values into their codewords and represent the set of codewords as a bit vector. Evaluating the predicate is then reduced to probing the bit vector. Note that technically, this is a special case of the previous type, but since it is non-trivial to express probing in a data-parallel way and an important building block for many predicates, we list it here as a separate type.

Arbitrary predicates: If none of the above optimizations can be applied, we provide a fall-back mechanism, where a block of codewords is unpacked as machine words into a buffer in cache, on which arbitrary predicates can be applied including arithmetic expressions and comparisons of several columns. This routine is also used to unpack a column for the subsequent database operator.

Figure 2 gives an overview about how the predicate evaluation works in our framework. As a rough overview, the following vectorized steps are performed on 4 to 256 codewords in parallel: In the first step, the data is brought into a format that the predicate can work on. In the most general case, this means *shuffling* all bytes containing bits of the codeword into the same machine word, *cleaning* the upper (unused) bits of the machine word, i.e., setting them to zero, *aligning* the codeword to machine word boundaries, i.e. shifting the machine word to the right, and finally storing the result into a *buffer*. In the second step, the predicate is evaluated. For the range scan, this consists of two *comparisons* and can be done directly after cleaning by shifting

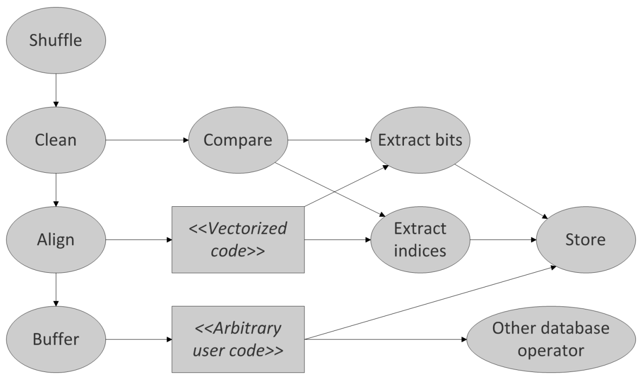

Figure 2: The four basic steps of all scans: (partially) unpack, evaluate predicate, extract result, and store result.

the range once before the scan. Vectorized predicates can be evaluated directly after the alignment to machine word boundaries, while we delay evaluation of other predicates until a buffer is filled in order to amortize virtual function calls etc. In the third step, the result of the predicate evaluation has to be *extracted*, be it a bit or an index indicating a match or the unpacked codewords in case of an arbitrary predicate. The last step is to *store* the extracted result.

In order to benefit fully of the SIMD instructions, many non-trivial optimizations have to be applied to this scheme. Skipping the alignment for range predicates and tightly interweaving unpacking and evaluation of vectorized predicates are two important optimizations already visible here. Additionally, we separately optimize every single bit case, i.e., we have a different implementation of the above scheme for every value of b. In some bit cases, a single special SIMD instruction can perform clean and align at the same time, while in others, shuffle alone needs three instructions. We have found reoccurring patters of how SIMD instructions can be combined intelligently and consider their description an important part of the contribution of this paper.

The rest of the paper is organized as follows: We first discuss related work in Section 2. In Section 3 we give details about how to efficiently implement our framework for scans with complex predicates. Section 4 presents and discusses the performance evaluation of these implementation. Finally, we provide our conclusion and outlook in the last section.

2. RELATED WORK

This sections relates this paper to existing work. We first review work on compression in database systems and on vectorized compression schemes outside the database world. Then we present previous work in the intersection of the two domains, i.e., vectorized database column scans, who all concentrated on scans with rather simple predicates.

2.1 Compression in Database Systems

The usage of compression has a long history in database systems. Of particular interest are those compression techniques that provide an efficient processing of the data in its compressed form.

Graefe and Shapiro [9] were among the first to study compression schemes that allow most of the query processing to be done on the compressed values. Roth et al. [21] and Westmann et al. [25] mention null suppression as compression technique for database systems. Lemke et al. showed in [17] that the compression of a column store can be used to overcome the memory wall.

2.2 Vectorized Compression in General

Independent of the usage in databases, Anh and Moffat propose their *Simple9* [4] and *Simple8* [5] compression schemes for posting lists. Both methods use the first 3 or 4 bits of each machine word as a selector, and the rest of the bits as a fixed number of fields with fixed length depending on the selector, leading to very fast decompression while maintaining a relatively good compression ratio. Schlegel et al. [22] implement and analyze vectorized compression schemes, among them a vectorized version of null suppression where only entire null bytes are suppressed and suppression is done for every word individually. Stephanov et al. [23] propose a SIMD variant of *varint* or *vbyte* based on the shuffle instruction. Wassenberg [24] uses a SIMD compression scheme for lossless image compression. Fang et al. [7] integrate several compression schemes into their GPU based query co-processor *GDB* to speed up queries by reducing the amount of data transferred to the GPU.

2.3 Vectorized Bit-Packed Compression and Scan

Several of the compression techniques in the previous section require a new data layout. This often results in optimizing either in terms of compression ratio, scan performance, or both. They are therefore very attractive and relevant for scan intensive databases as SAP HANA database. On the other hand, one of the main benefits of bit-packed compression is the ability to perform random access in constant time and small overhead, because of its fixed-size encoding. Lemke et al. [17] also use it as a basic building block of their compression framework. The standard bit-packed compression is therefore still the most versatile method in the general purpose system SAP HANA database and many authors have worked on optimizing it.

Lamport [14] was the first to pack several code words into a single machine word and perform arithmetic and logic operations in the compressed space. To make some of these operations simpler, the author adds an additional separation bit.

Later, unpacking of bit-packed data was optimized by Zuchowski et al. [27,28] in the context of domain and dictionary encoding in their column-store data base system. By extensive loop-unrolling and implementing individual routines for each number of bits, the offsets and masks turned into loop invariants, which can be stored as constants. Please note that they use the term *vectorization* in a different way and their implementation does not feature SIMD instructions. They also proposed the very popular compression scheme *PFOR-Delta* (*Patched Frame-of-Reference Delta*), which has the bit-compression unpacking as a subroutine.

Blink's [20] *Frequency Partitioning* divides the table into partitions based on the frequency of tuples and applies fixed length bit compression to every partition. Their codes also have a separator bit so that the scan can evaluate inequalities predicates using SIMD instructions without machine word alignment. They do not mention more complex predicates.

Apart from this, *Bit Packing* was reimplemented and confirmed by many authors in the context of column store databases as a special kind of *Null Suppression* suited for dictionary compression [3, 10].

Holloway et al. [11] investigate on the implications of compression on a full table scan. Like us, they see the increased importance of full table scans, but their setting is slightly different: they combine decompressing blocks, (groups of) columns, and fields with predicate evaluation into a single generated piece of C code. Furthermore, they also make use of interleaving the decompression of multiple blocks, in order to improve instruction level parallelism and instruction flow.

Their bit-vertical format *BitWeaving/V* (first proposed by O'Neil et al. [19]) decomposes codes bit-wise to different locations, which results in a significant advantage for scan performance through pruning, but puts a huge burden on any other operation than scanning due to the higher tuple reconstruction costs. This works well for predicates looking once at each bit and in their natural order, which is the case for inequalities and ranges, but does not seem to be applicable to more complex predicates. Their bit-horizontal format *BitWeaving/H* is similar to the SIMD-scan, with the following differences: (1) code words are separated by an additional delimiter bit (like Blink [20]), (2) decoding uses only full-word instructions (as opposed to SIMD instructions and unlike Blink), and (3) code words do not span machine words, i.e., machine words are padded if they are not entirely filled by code words. This reduces the compression ratio, especially for code words of 21 bits or more where it results in the same size as an uncompressed vector.

Last but not least, Lemire and Boytsov [15] survey the performance of a large variety of data parallel integer compression schemes for posting lists. The fastest approach in their survey (*SIMD-BP128**) is based on bit-packing, which is very similar to ours. They also analyze bit-packing in isolation, comparing horizontal bit packing with SIMD, vertical bit packing with SIMD and manually optimized bit packing without SIMD. They also find that a vertical layout is faster for unpacking than a horizontal layout. Finally, they foresee the advantage of wider bit width and more powerful instructions of Intel AVX2 as it is presented in this paper. As their implementation is publicly available [16], Section 4 contains a direct comparison between their and our implementation.

3. ALGORITHMS

This section provides a detailed description of the different parts of our scanning framework as well as many of the optimization needed for a performant implementation. We start with the description of the (partial) unpacking of codewords. We then show how range predicates can be particularly optimized and how the result is either stored as a bit vector or an index vector. Finally, we show how the In-List predicate can be vectorized.

3.1 Bit-Compression Unpacking

We will first focus on decoding codewords from bit-compression into 32-bit integers. Since it is normally the first step of a scan before the evaluation of a predicate, we examine the unpacking separately. Subsequent steps of the scan can then

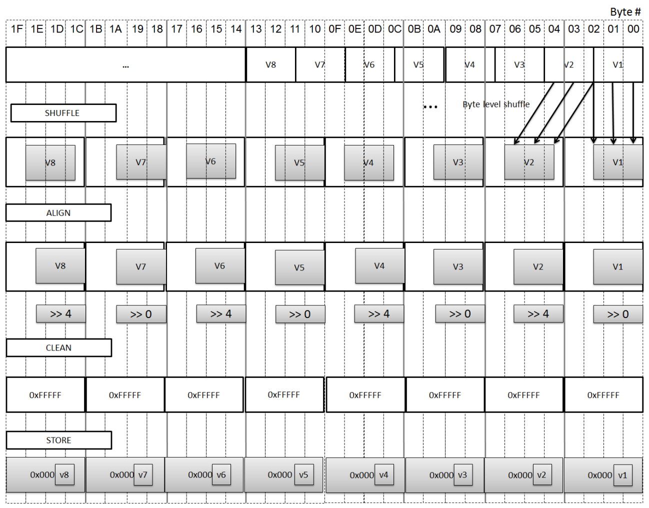
Byte #
1F 1E 1D 1C 1B 1A 19 18 17 16 15 14 13 12 11 10 0F 0E 0D 0C 0B 0A 09 08 07 06 05 04 03 02 01 00
V8 V7 V6 V5 V4 V3 V2 V1
SHUFFLE
Byte level shuffle
ALIGN
>> 4
>> 0
CLEAN
0xFFFFF
STORE
0x000

Figure 3: Core steps of the unpacking algorithm on a 20 bit-case example.

either (1) directly evaluate the predicate using SIMD instructions or (2) store the result in a buffer for processing it sequentially.

Figure 3 illustrates the core steps of the unpacking algorithm in the case of 20 bits per codeword as an example. The basic ideas apply to all bit cases, but may change in detail. The upper line shows the packed codewords with 20 bits each. Those codewords are distributed across the vector register using a shuffle instruction for bytes, so that each double-word contains all bits of one codeword. Please note that some of the bytes need to be replicated into two double-words as they contain bits from two different codewords. In the third line, the codewords are then shifted to the right in order to align them. As a last step, a mask is applied to clean all other bits, which results in one codeword per double-word.

Algorithm 1 Unpacking algorithm

```
 1: set $k$ to 0
 2: for $i$ from 0 to $max\_index/128$ do
 3:    for $j$ from 0 to 15 do
 4:       parallel_load $v$ from input$[k * 16 + j * b]$
 5:       shuffle $v$ using shuffle_mask$(m_0, ..., m_{31})$
 6:       parallel_shift $v$ by $(s_0, ..., s_7)$
 7:       parallel_and $v$ by $(a_0, ..., a_7)$
 8:       parallel_store $v$ in output$[i * 16 + j * 8]$
 9:    end for
10:    increase $k$ by $b$
11: end for
```

The same scheme for unpacking codewords from an array of bit-fields is given as Algorithm 1. `input` is a pointer to a byte array and `output` is a pointer to an integer array. Only the first codeword is aligned to a byte boundary. However, since we use bit-fields of fixed size, this repeats after eight codewords. As we process eight codewords in parallel, the loading line 4 has always values that are aligned to bytes. Lines 5 and 6 then realize the shuffling and shifting of the entries. Again, the pattern for shuffling and shifting repeats

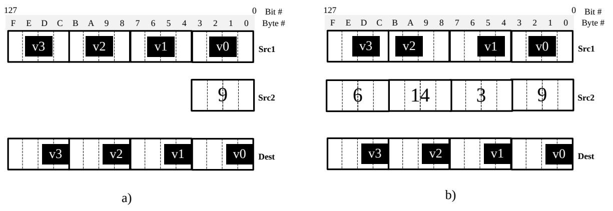

Figure 4: (a) With Intel SSE2 all vector elements are shifted by the same amount. (b) With Intel AVX2 element of the vector register can be shifted independently.

after eight entries and the arguments m and s of the respective instructions are therefore constant vectors. Finally, the result can be cleaned and stored in line 7 and 8. Instead of a simple loop, we use a nested loop where the inner loop processes 16 codewords. If you then completely unroll the inner loop, $j * b$ turns into a loop invariant and does not need to be computed at run-time.

This algorithm almost directly translates to Intel AVX2 instructions. In particular, there are some notable simplification compared to our SSE-based implementation [26]:

- By processing 8 instead of 4 codewords in each iteration, only one constant is needed for the shuffle mask and the shift offsets, respectively.
- By using unaligned loads, it is possible to load registers in a way that the first codeword always starts at a byte boundary. This not only results in simpler code but turned out to be faster, as current architectures are much more forgiving to unaligned loads. There is still a performance penalty when data loads are split across cache lines, but this penalty is amortized by the reduction in the other cases. As Intel AVX2 organizes the data in lanes of 128 bit, it is not possible to shuffle bytes across the lane boundary. However, this can be simply overcome by loading the lower and the upper lanes independently. Since the Haswell architecture is equipped with two load ports, this does not impose any performance disadvantage.
- Instructions for shifting values in a SIMD register were already available in previous generations. However, theses instructions were always shifting all elements by the same shift amount (see Figure 4 left). In our SSE-based implementation, a multiplication was used to workaround this limitation. With Intel AVX2, Intel has introduced a new variant of the shift instruction for this and similar situations. The new vector-vector shift instructions (`vpsrlvd, vpsrlvq, vpsravd, vpsllvd, pvsllvq`) allow another SIMD register as second argument, which determines for each element the number of bits that the corresponding element in the first argument is shifted.

The standard algorithm processes 8 codewords per iteration. For lower bit-cases, it is possible to process 16 or 32 codewords per iteration by using instructions that operate on data types of size byte, word, and double-word. Apart from this, many other optimizations are possible for specific

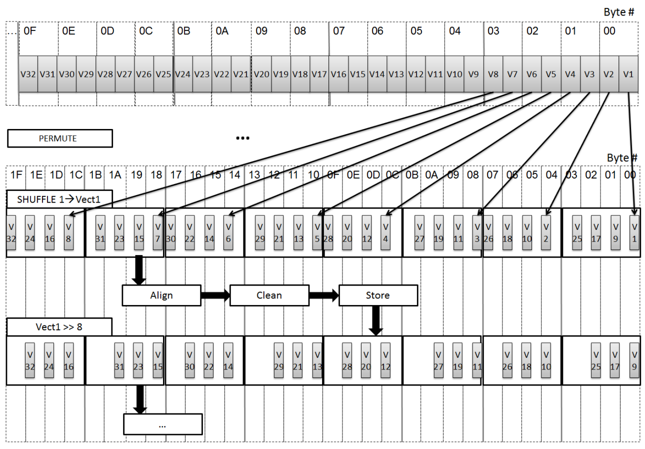

Figure 5: Permutation trick to unpack 32 codewords with one load for bit-cases 2 and 4.

bit-cases. Depending on the bit-case, we apply one of more of the following optimizations:

Unpack32 is an optimization that allows to decompress 32 codewords in one iteration with a single load. This optimization applies for bit-cases 1, 2, 4, and 8. When a parallel load is executed, the resulting 256-bit register contains 32 or more codewords for lower bit-cases. Then, we shuffle in a way to have a codeword in every byte. By ordering the codewords appropriately we can extract 8 consecutive codewords at a time. Figure 5 is an example of Unpack32 for bit-case 4.

Unpack16 is a similar optimization that applies to bit-cases 3, 5, 6, 7, 9, 10, and 16 and allows extracting 16 codewords in each iteration. The difference is that after a load, we shuffle in a way to have one codeword per word. The order of the shuffled codewords follows the same logic as for Unpack32.

NoAlign removes the shift instruction for the alignment in bit-cases 8, 16, 24, and 32, as the data is already byte-aligned.

16cvt32 is an optimization that replaces the three instructions, shuffle, align, and clean by one conversion instruction for bit-case 16: *vpmovzxwd*. This instruction converts a 128-bit vector with eight 16-bit integers to a 256-bit vector with eight 32-bit integers, thus zeroing the upper bytes.

Broadcast: For the lower bit-cases, it is possible to save one load by loading an SSE vector and broadcasting it to the upper lane with the *vbroadcasti128* instruction.

OptimizeAlign is an optimization that, combined with the appropriate shuffle masks, allows to align the data with an optimized number of instructions ranging from one independent shift for bit-cases 3, 5, 6, 7, 9, and 10 to three independent shifts for bit-case 30.

3.2 Range Predicate with Index Vector Result

A common operation of data is the search of entries that fulfill a certain criterion. In this section, we therefore describe how the data unpacking and filtering can be combined for achieving better performance than two individual steps.

The first variant of the scan algorithm handles range search operations with very low selectivity. It takes a range *min* and *max* as a predicate and yields an index vector that contains the indices of the codewords that are within the range. The general scheme is listed as Algorithm 2 and follows the same outline as was presented in our earlier work.

Algorithm 2 Scan with range predicate returning index vector

```
 1: parallel_shift ($min, min, min, min, min, min, min, min$)
    by ($s_0, ..., s_7$), store in $min$
 2: parallel_shift ($max, max, max, max, max, max, max, max$)
    by ($s_0, ..., s_7$), store in $max$
 3: set k to 0
 4: for $i$ from 0 to $max\_index/128$ do
 5:    for $j$ from 0 to 15 do
 6:       parallel_load $v$ from input[k*16 + j*b]
 7:       pshuffle $v$ using shuffle_mask($m_0, ..., m_{31}$)
 8:       parallel_compare $v$ with ($min$, $max$), store in $t$
 9:       convert $t$ to a vector of hit indices $h$
10:       store $h$ in output[$i * 16 + j$]
11:    end for
12:    increase $k$ by $b$
13: end for
```

Lines 4 to 7 correspond to the first part of the unpacking algorithm 1. However, the shifting inside the loop can be avoided by shifting the lower and upper limit *min* and *max* before the loop (lines 1 and 2), and then comparing the shifted range in line 8. Unfortunately, step 9 cannot be implemented with Intel AVX2. We therefore extract and store the indices individually.

Similarly to the unpacking algorithm, **NoAlign**, **OptimizeAlign**, and **Broadcast** optimizations apply. Nevertheless, the shuffle and shift masks are different, as the only restriction we have for alignment is the fact that the comparison instructions are designed for integers while we use unsigned integers. Therefore, it is important for the correctness of the comparisons to always keep the most significant bit of a double-word to zero. In other words, the shuffle operation must avoid the most significant bit for the evaluation of an arithmetic predicate.

Moreover, this algorithm allows for another optimization:

TestZ uses *vptest* instruction to test whether the comparison result is null (i.e., no hits). If the test succeeds, it continues to the next iteration without executing line 9 of the algorithm. Since this variant of the scan is typically used for searches with low selectivity, this situation often occurs in practice.

3.3 Range Predicate with Bit Vector Result

The second variant of the scan algorithm handles the common operation of a range search returning the result in form of a bit-vector. It takes a range *min* and *max* as a predicate and yields a bit-vector where bits set to 1 are hits and the bit position corresponds to the index position of the codeword that was hit. The general scheme follows the same

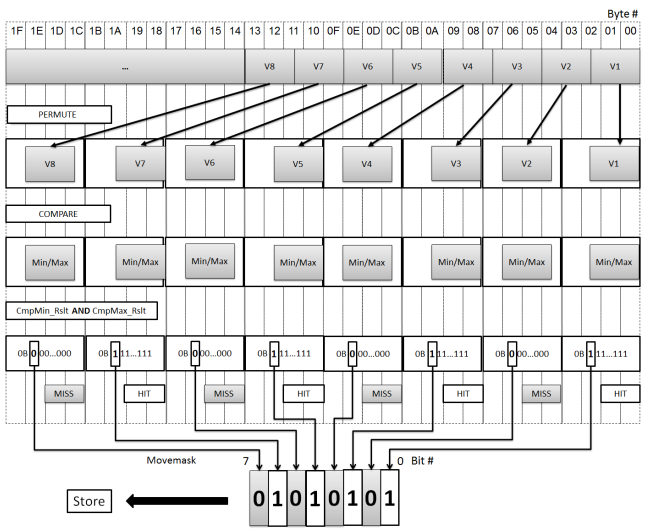

Figure 6: Core steps of the range scan algorithm on a 20 bit-case example.

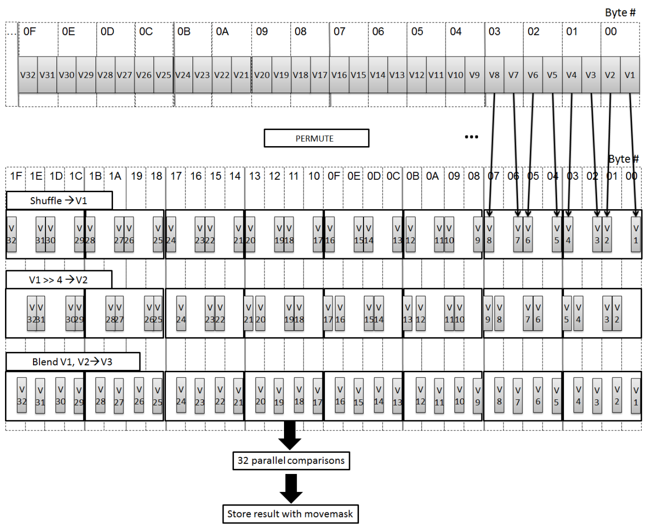

Figure 7: Permutation trick to perform 32 parallel comparisons for bit-cases 2 and 4.

outline as was presented in our earlier work [26]: A formal description is given as Algorithm 3. The only difference to Algorithm 2 is the processing of the result. Instead of a list of indexes, a bit-vector must be generated. The previous steps of shuffling the entries and evaluating the range predicate stay the same. Figure 6 illustrates the construction of a bit-vector as the last step at the example of bit case 20. Again, the same ideas apply to the other bitcases.

Algorithm 3 Range scan with bit vector result

```
1: parallel_shift $(min, min, min, min, min, min, min, min)$
   by $(s_0, ..., s_7)$, store in $min$
2: parallel_shift $(max, max, max, max, max, max, max, max)$
   by $(s_0, ..., s_7)$, store in $max$
3: set $k$ to 0
4: for $i$ from 0 to $max\_index/128$ do
5:     for $j$ from 0 to 15 do
6:         parallel_load $v$ from input$[k * 16 + j * b]$
7:         shuffle $v$ using shuffle_mask$(m_0, ..., m_{31})$
8:         parallel_compare $v$ with $(min, max)$, store in $t$
9:         convert $t$ to 8-bit integer $r$
10:        store $r$ in output$[i * 16 + j]$
11:    end for
12:    increase $k$ by $b$
13: end for
```

Similarly to the decompression algorithm, the optimization of individual bit-cases can result in significant performance improvements. **NoAlign**, **OptimizeAlign**, and **Broadcast** introduced earlier for the decompression algorithm also apply for the scan with range predicate algorithm, but with different masks and less instructions, as we do not need to byte-align the data. In addition, the following optimizations apply:

Movemask: Step 9 can be executed with one instruction that creates a mask from the most significant bit of each 32-bit element in the AVX2 vector: *vmovmskps*.

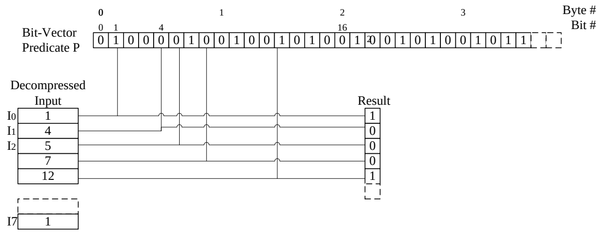

Figure 8: When an In-List predicate is given as a bit-vector P, the scan operation must check for each decompressed codeword t if the bit $P[t]$ is set.

LogicCmp is an optimization for bit-case 1. It takes advantage of the fact that in bit-case 1, we have only four possible ranges. Lines 7 to 10 are then effectively replaced by either setting all bits to zero, to one, to the bits of the column, or to the negated bits of the column.

Scan32 enables 32 parallel comparisons for bit-cases 2 and 4. With a shuffle, a shift, and a blend, we move 32 codewords, one in each byte of the 256-bit vector. We keep the initial order of the codewords. We use the *vpcmpgtb* instruction to compare packed 8-bit integers. Besides, we use *vpmovmskb* instruction to convert the result to a 32-bit integer instead of an 8-bit. Figure 7 is an example of how to set the data to enable 32 parallel comparisons for bit-case 4.

Scan16: Similarly to Scan32, the optimization enables 16 parallel comparisons for bit-cases 3 to 10 and 12.

3.4 In-List Predicates

This is the most general variant of the scan algorithm where the condition of the search is an arbitrary subset of the codeword domain. The in-list is represented as a bit-vector where each bit corresponds to the value of its index. A bit is set to 1 iff it is searched. In the following, we present the algorithm producing a bit vector as result, but the same ideas apply for the algorithm producing an index vector. The outline of the scalar algorithm is as follows:

1. Compute the address of double-word that contains the bit as $addr := base + pos/32$.
2. Load double-word B from memory at $addr$.
3. Compute the bit position inside B as $p := pos\%32$.
4. Extract bit b from B by shifting B to the left by $31 - p$ bits and extracting the sign bit.

Since the second argument of the integer division and remainder is a power of 2, it can be implemented efficiently by a *shift*ing by 5 and an *and*ing with 31, which is a standard optimization for compilers. With these modifications, the evaluation of the In-List predicate directly translates to the pseudo code that is used in Algorithm 4.

Algorithm 4 Scan algorithm with In-List predicate

```
 1: set $k$ to 0
 2: for $i$ from 0 to $max\_index/128$ do
 3:   for $j$ from 0 to 15 do
 4:     parallel_load $v$ from input$[k * 16 + j * b]$
 5:     shuffle $v$ using shuffle_mask$(m_0, ..., m_{31})$
 6:     parallel_shift $v$ by $(s_0, ..., s_7)$
 7:     parallel_and $v$ by $(a_0, ..., a_7)$
 8:     parallel_shift $v$ by $(5, ..., 5)$, store in $offset$
 9:     gather elements from $P$ with indices $offset$, store
        in $B$
10:     parallel_and $v$ by $(31, ..., 31)$, store in $p$
11:     parallel_substract $p$ from 31, store in $p$
12:     parallel_shift $B$ left by $p$ bits
13:     convert $B$ to 8-bit integer $r$
14:     store $r$ in output$[i * 16 + j]$
15:   end for
16:   increase $k$ by $b$
17: end for
```

Apart from lines 8-12, the algorithm coincides with the range scan in Algorithm 3. Lines 8-12 replace the evaluation of the range predicate with the evaluation of the In-List predicate as follows: In line 8, we compute the offset of the address $pos/64$ as well as the shift values $n := 31 - pos\%64$ in lines 10-12. The loading from the bit-vector in line 9 requires some extra handling in form of a *gather* instruction. Normal load instructions for a vector register load a consecutive piece of memory into the register. In other words, the data elements of the vector are filled with numbers that reside next to each other from memory. This matches perfectly the usage when data is processed from an array or matrix. However, it poses a severe problem for vectorizing table look-ups. For these scenarios, Intel AVX2 offers a *gather* instructions, where each element is loaded according to an index in a table.

Similarly to the previous algorithms, bit-case level optimizations represent a significant performance boost. For instance:

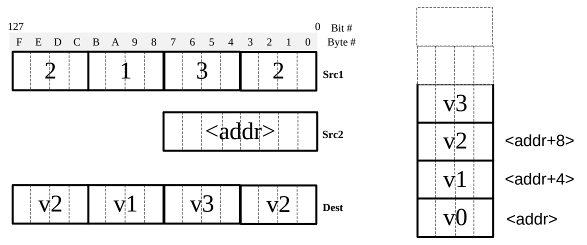

Figure 9: The *gather* instructions loads elements from memory based on a base address and offsets for each data element.

Permute replaces line 10 by a permute (*vpermd*) instruction on a 256-bit register that contains the bit-vector predicate. This is true for the lower bit cases where the predicate is not larger than 256 bits. In particular, we use this optimization for bit cases 6, 7, and 8. We also apply a similar optimization to bit cases 9 and 10, where the predicate fits respectively into two and four AVX vectors.

AvoidGather is illustrated in figure 10. It is applicable to bit cases 2, 3, 4, and 5. This optimization relies on the fact that the bit vector predicate fits into a 32-bit word. Therefore, we set each 32-bit word of an AVX vector to the value of the predicate. Then, we convert each codeword from the loaded segment to a 32-bit vector where only one bit is set to 1. This bit's index corresponds to the initial codeword. We execute a logical *and* on the constant and the converted vector. We then compare equality between the resulting vector and the converted vector which gives us the final result.

3.5 Summary

Table 1 summarizes which optimizations are used for every bit case.

Bit-case	Unpack	Scan Range-to-BV	Scan Range-to-IV	Scan BV-to-BV
1	★	♣ #	♠	♣ ▽ #
2	★ □ ▼	▲ □ ▼ #	▲ □ ▼ ♠	▲ □ ▼ ▽ #
3	♦ □ ▼	◊ □ ▼ #	◊ □ ▼ ♠	◊ □ ▼ ▽ #
4	★ □ ▼	▲ □ ▼ #	▲ □ ▼ ♠	▲ □ ▼ ▽ #
5	♦ □ ▼	◊ □ ▼ #	◊ □ ▼ ♠	◊ □ ▼ ▽ #
6 and 7	♦ □ ▼	◊ □ ▼ #	◊ □ ▼ ♠	◊ □ ▼ ∞ #
8	♦ △	◊ □ △ #	◊ □ ♠	◊ □ △ ∞ #
9 and 10	♦ ▼	◊ ▼ #	◊ ▼ ♠	◊ ▼ ∞ #
11	▼	▼ #	▼ ♠	▼ #
12	♦ ▼	◊ ▼ #	◊ ▼ ♠	◊ ▼ #
13	▼	▼ #	▼ ♠	▼ #
14	▼	▼ #	▼ ♠	▼ #
15	▼	▼ #	▼ ♠	▼ #
16	♦ ■ △	□ △ #	□ △ ♠	□ △ #
17 to 23	▼	▼ #	▼ ♠	▼ #
24	△	△ #	△ ♠	△ #
25 to 31	▼	▼ #	▼ ♠	▼ #
32	△	△ #	△ ♠	△ #

★ Unpack32	♦ Unpack16	■ 16cvt32	♠ TestZ	▲ Scan32
◊ Scan16	□ Broadcast	♣ LogicCmp	△ NoAlign	▼ OptimizeAlign
▽ AvoidGather	∞ Permute	# Movemask		

Table 1: Summary of optimizations

4. EVALUATION

We now measure the scan performance with different predicates and, where available, compare it with the performance of implementations of previous publications.

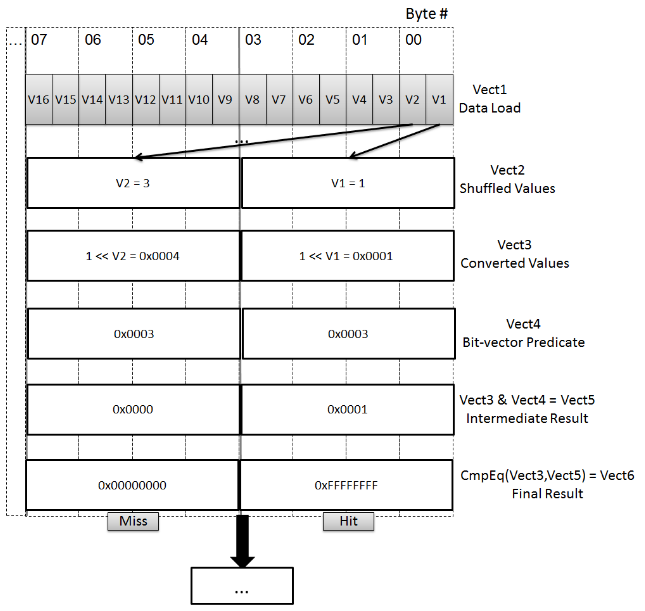

Figure 10: Avoiding the Gather instruction for lower bit-cases. Illustrated example for bit-case 4.

Unless otherwise mentioned, all experiment were run on an Intel© Core™ i5-4670T processor [12] (based on the architecture codenamed Haswell) with a nominal clock speed of 2.3 GHz, a maximal Turbo Frequency of 3.3 GHz, four cores, and a 6MB of L3 cache. Our machine was equipped with 6GB of main memory and runs on SLES 11.1 using a Linux kernel 3.6.0-rc7. We use Intel Composer XE 2013 update 1 (ICC) as compiler for our routines and GCC 4.7.3 (GCC) for the others, which were the fastest compilers for the respective routines [1]. For the implementation of our algorithms, we used intrinsics. This has the advantage we get the full control over the instruction that are used, but tedious tasks like register assignment or instruction reordering is left to the compiler.

We run our test on data that we generate uniformly at random in order to model a hard scenario for routines depending on the data. The only place where this is the case is where the index vector result is produced. All other routines are completely free of conditional jumps and hence do not depend on any data skew. In order to change the selectivity of the range predicates, we change just the probability of the highest relevant bit and run the scan with the range predicate $[0, 2^{bit-1})$. We always give averages of 10 runs, each processing 2^{28} codewords; if not otherwise mentioned, a run consists of processing the same 2^{25} codewords 8 times. We use Intel Performance Counter Monitor (Intel PCM) 2.5 [6] to measure cycles and nanoseconds and observe that the CPU always runs at its maximal frequency of 3.3 GHz. Since the two metrics have a constant ratio of 3.3, our figures have two axes and show nanoseconds and cycles in one plot.

[1]In other experiments shown in Appendix A, Lemire compiled with ICC was roughly 5% slower and AVX2-Scan roughly 30% faster than compiled with GCC. SIMD-Scan had the same performance with either one.

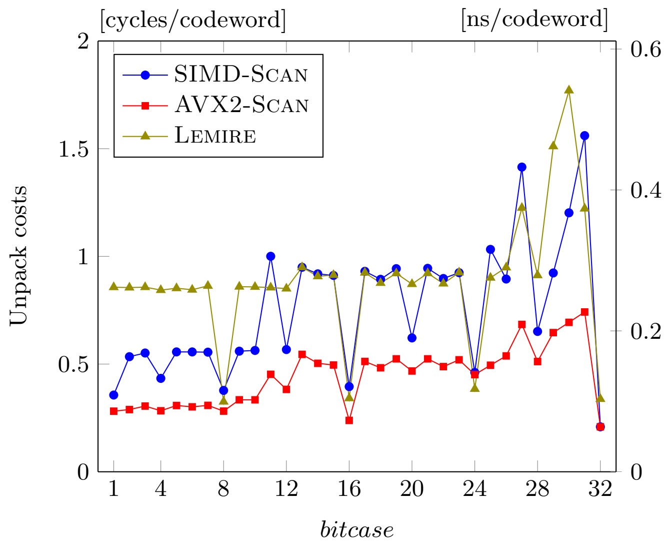

Figure 11: Isolated unpacking costs of different implementations.

In the following, we analyze the performance of the range predicate scan with both index vector and bit vector result. This is the most common predicate and we can compare it with previous work. Furthermore, we can also compare the impact of the output data format. As a particularly hard example of vectorizable predicates, we also show experiments with the In-List predicate as the most general stateless predicate, but argue that equivalent or better performance is easily achievable for simpler predicates. We also show the pure unpack performance of our fall-back mechanism for arbitrary predicates, which represent the base cost of such a scan. Depending on the user-written predicate code, additional costs applies obviously.

Each of these predicates is embedded in up to four scan implementations: our previous SIMD-Scan [26] with additional optimizations based on SSE4, our new AVX2-Scan based on the new Intel AVX2, Lemire, a reimplementation of SIMD-Scan by Lemire and Boytsov [15] based on SSE4, and Cache Ping. Cache Ping is a set of generated C functions, one predicate type and bit case, "pinging" the cache lines that any algorithms has to read from and write to. It does not evaluate any predicate or even unpack the codewords but preserves data dependencies (with cache line granularity). Since it does the minimal data movement of every algorithm, it models the memory bound part of scanning and unpacking and serves as a comparison.

4.1 Unpacking Codewords

We start our analysis with completely unpacking small arrays of 1024 codewords in order to compare the computational costs of unpacking. Both input and output reside entirely in the fastest cache level, eliminating thus the effect of potentially slow memory. The results are shown in Figure 11. As a first observation, the three routines SIMD-Scan, AVX2-Scan, and Lemire can unpack almost all bit cases in less than a cycle per codeword, but the exact number depends highly on the bit case. This is a consequence of the fact, that every bit case is processed by its own algo-

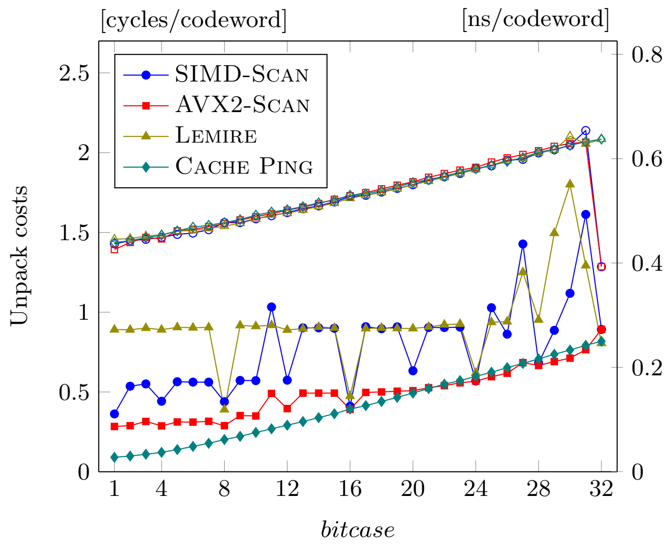

Figure 12: Unpacking costs of a full column-scan *without* the evaluation of arbitrary user code, into a buffer (solid markers) and into RAM (hollow markers).

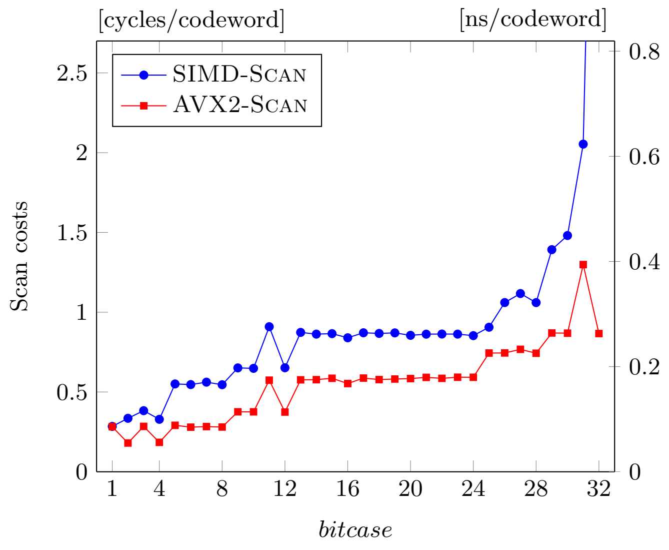

Figure 13: Scanning costs with range predicate producing an index vector for low selectivity ($s = 1/2^{14}$).

rithm, and the respective sets of optimizations applied are different. In particular the bit cases that are multiples of eight are much easier to unpack, which results in considerably lower unpacking time per codeword. AVX2-Scan is consistently around 30% faster than SIMD-Scan and has less hard cases, needing just about half a cycle for most bit cases, since the more powerful instruction set gives the opportunity for more and faster optimizations.

The experiments also show that Lemire's implementation is reaching almost equivalent performance. Still, in 14 out of 32 bit cases, our updated routines are around 30% faster. On the other hand, their implementation surpasses our performance in 5 bit cases. This difference in the two implementations shows the big influence of a careful implementation. In particular, the performance numbers in this paper also reflect some optimizations of the implementation of SIMD-Scan compared to our previous results [26].

4.2 Buffered Unpacking for Arbitrary Predicates

We now analyze to what extent it is possible to achieve this unpacking performance during a scan. To this aim, we run a fall-back scan designed for arbitrary predicates with an *empty* predicate, hence measuring the costs for memory transfer and unpacking. We use a buffer size of 1024 codewords, since this is the choice made in SAP HANA database. We also experimented with other buffer sizes and confirm that the current value provides a good trade-off between amortizing the costs for changing execution context between unpacking and further processing and a small cache footprint for the buffer, in particular if several columns are unpacked at the same time.

The respective lower lines in Figure 12 show that SIMD-Scan and Lemire have the same unpacking costs on the entire column as in the pure in-cache scenario. This is also true for AVX2-Scan for bit cases up to about 16. In these cases, the computations dominate the memory transfer costs. For bit cases higher than 16, AVX2-Scan starts being memory bound, again emphasised by the fact that it has the same performance as just pinging the same cache lines. This is a hint that with current hardware, we should not use more expensive data formats with higher compression.

The respective upper line of each algorithm in Figure 12 shows the costs of *unbuffered* unpacking, i.e., where the unpacked column is materialized into RAM: they are considerably higher and exactly the same for all algorithms, since they are all dominated by the memory transfer costs. This argument is strengthened by the fact that not even pinging the cache lines is faster (see algorithm Cache Ping). In another experiment not presented here, we found out that using non-temporal stores for writing directly into RAM without going through the cache can lower the costs by roughly factor two, but still leave the routines memory bound. The `memcpy` used in bit case 32 of our routines achieves this performance already.

4.3 Range Predicates

Next, we show the performance of a scan with range predicate where an index vector of matching rows is produced. The costs per codeword depend heavily on the selectivity of the scan, since only indices of selected rows need to be computed and written to memory. Figure 13 shows the costs of a scan with low selectivity ($s = 1/2^{14}$), which is what this scan variant is built for. For such a low selectivity, the performance is very similar to unpacking: less than one cycle per codeword for most bit cases for SIMD-Scan and roughly 30% less for AVX2-Scan. Bit case 32 of SIMD-Scan is in fact a scalar implementation, since without the missing signed comparison in SSE, we would have to use 64 bit types, which operates on two times less codewords per instruction. Both SIMD-Scan and AVX2-Scan would benefit from a signed comparison instruction.

Figure 14 shows our experiments for the scan with range

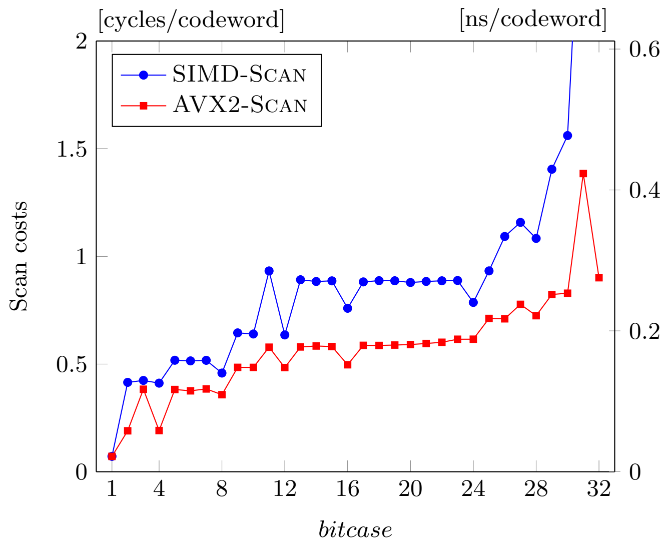

Figure 14: Scanning costs with range predicate producing a bit vector.

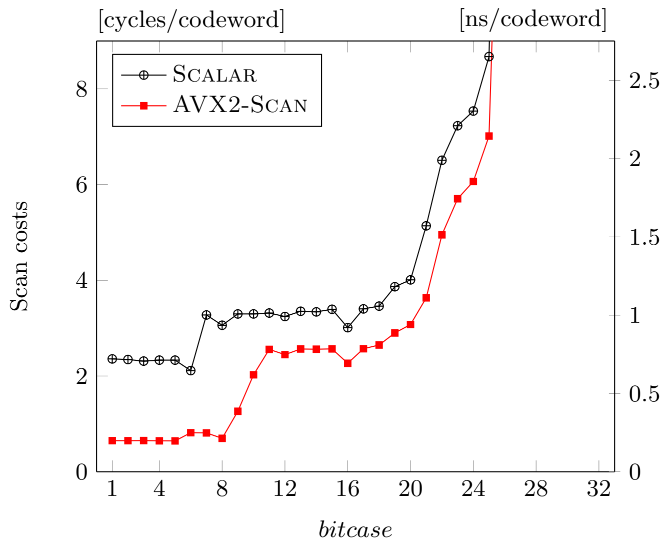

Figure 15: Scanning costs with In-List predicate producing a bit vector.

predicate where a bit vector is produced as result. Unlike the scan variant producing an index vector, its output size does not depend on the number of matches, so its performance is completely independent of the selectivity. As a consequence, the performance is very similar to unpacking: all bit cases have different costs, but most of them remain below 1 cycle per codeword, and AVX2-Scan has roughly 30% lower costs than SIMD-Scan. Notice also that some bit cases have lower costs than unpacking, since some of the steps of unpacking can be merged or skipped.

These results contradict the findings of Li and Patel [18], who also reimplemented our first version of SIMD-Scan. In the absence of availability of their reimplementation, we can only compare the numbers they report with the numbers we reported [26] with the same predicate and on slightly older hardware: While their reimplementation costs constantly 5 cycles per codeword, we needed around 1 cycle/codeword for unpacking at the time [1]. Two indications let us believe that a suboptimal reimplementation is the explanation for the performance difference of factor 5: First, manually optimizing every bit case separately by intelligently combining SIMD instructions should lead to different costs for each bit case. Second, they write that they scan exactly four codewords at a time, independently of the bit case, while we showed in Section 3 that even in SIMD-Scan, we can scan 4, 8, 16, and even 128 comparisons at a time using smaller data types, and double that in AVX2-Scan. We see this as another argument for the importance of the optimizations described earlier.

[1]This was not explicitly stated in the previous paper, but could be deduced from the plots: Figure 11 gave an unpack time of around 400ms for 1B codewords in most bit cases, corresponding to 0.4ns/codeword. With a frequency of less than 3.0GHz, this results in less than 1.2 cycles/codeword. As mentioned at the time and as visible in our new experiments, unpacking and range scan have very comparable performances.

Their first alternative, BitWeaving/V, needs around 1 cycle per codeword for bit cases higher than 16 and as little as to 0.1 cycles for lower bit cases and is therefor somewhat faster than our SIMD-Scan. Their BitWeaving/H, which as opposed to the other method also supports fast single look-up, seems to have a very similar performance than our SIMD-Scan or slightly worse in the higher bit cases.

4.4 In-List Predicates

Last but not least, we analyze the performance of a scan with In-List predicate. The results of our experiments are shown in Figure 15. We compare AVX2-Scan with a scalar version, since we were not able to produce a faster version based on SSE4. With the In-List predicate, the performance depends on the size of the bit vector (which is 2^b for bit case b) and also on the data distribution of the column, since there is a look-up into the predicate for each row. Since we run on uniformly distributed data, we expect the plotted costs to be close to an upper bound. For bit cases higher than $b = 21$ and $b = 26$, the predicate is of size of 2^bbit $= 2^{b-3}$byte, which is larger than the L2 and L3 cache of our machine respectively. As a consequence, the impact of cache misses start to dominate to performance.

For the more relevant lower bit cases, the performance is very appealing: Scalar can scan a codeword in less than 4 cycles and AVX2-Scan in less than 3. Thanks to the vector-vector shift and gather instructions, AVX2-Scan even needs less than 1 cycle for bit cases 1 to 8.

4.5 Summary

We now summarize our evaluation with a comparison of the throughput of the scans presented above. In order to group the performance of all bit cases of a certain scan, we aggregate them into a box plot presented in Figure 16. In this plot, the thick line inside of each box indicates the median, the box itself indicates the first and third quartile, and the whiskers indicate the minimum and maximum value of the respective groups. As seen in the detailed analysis

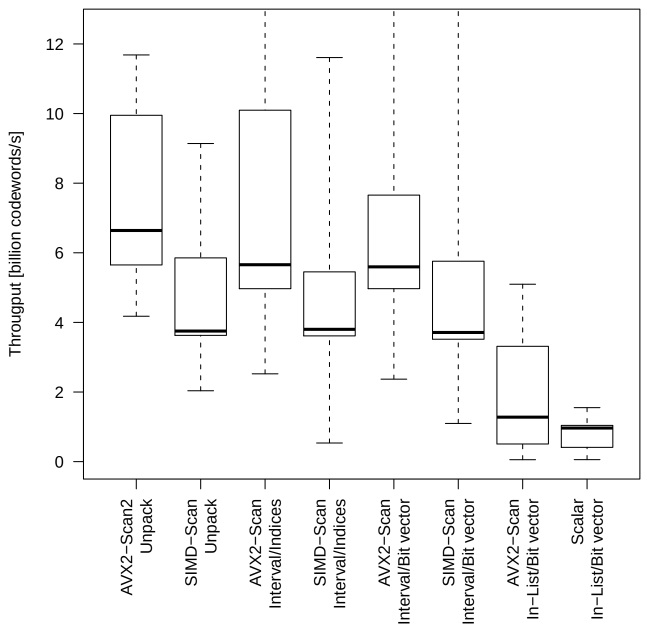

Figure 16: Scan throughput for different predicates, output formats, and implementations. Each box regroups the 32 bitcases of a variant.

before, most bit cases lie in the same ballpark, but some of them are worse (usually high bit cases) and some of them are better (small bit cases).

For our fall-back scan (with empty predicate, i.e., unpacking) and for scans with range predicate producing either output type, AVX2-Scan achieves a throughput of 6-8 billion codewords/s for most bit cases. Unpack has a peak performance of 10-11 billion codewords/s for bit cases 1 to 8 and scanning even above 17 billion codewords/s for bit cases 1, 2, and 4. The scan with more complex predicates have also a competitive performance: we can scan 1-3 billion codewords/s with a bit-vector predicate. These performances are roughly 30% higher than the respective equivalents of SIMD-Scan, except for the scan with bit vector predicate, where the improvement is even higher.

5. CONCLUSION AND OUTLOOK

We have presented a framework for the vectorized evaluation of complex predicates in database column scans. We have classified scan predicates into categories of different complexity and have shown how to bring their evaluation as close as possible to the compressed data. We gave many details about the vectorization of unpacking and result extraction with Intel AVX2, which are essential to improve our implementation compared to prior work. In particular, we presented a way to vectorize scan operations like the scan with bit-vector predicate that seemed to exhibit only little data parallelism. We conclude that our scan framework, in particular the AVX2-Scan implementation, is an effective means to improve scan performance. Since we do not change the underlying data format, no other database component is affected; in particular, fast single lookup performance is not compromised.

We are expecting hardware vendors to further increase the vector width and expressiveness of their instruction sets, which might allow faster or higher compressed data formats. In particular, Intel has recently released the description of Intel AVX-512 [2]. In the context of bit-fields unpacking and scans, the most notable additions are the following:

512bits: The register length will be extended to 512 bits. All of the algorithms described in Section 3 can naturally extended to the wider registers.

Mask registers will be added to allow the masking in vector registers to certain elements. Furthermore, the mask registers will serve as the destination for vector comparisons. This simplifies the processing of bit-vectors. In particular, the *and* operation for the interval predicate as well as the *movmask* instruction for bit-vector output can be completely eliminated.

Cross-lane shuffles: Instead of loading lanes individually, it will be possible to use a single load and a shuffle.

Compress instruction: The *compress* instruction allows to storing the a sub-set of the vector elements by a given mask. Please note that this differs from a masked move, where the elements are blended with the target. As a consequence, the *compress* instructions allows an efficient way to store the result of a scan as a list of hits or indexes.

Unsigned comparison: The scan with interval predicate requires some extra care with Intel AVX2 if the most-significant bit is used. Using the unsigned comparison in Intel AVX-256, this special handling can be avoided.

Intel AVX-512 will be first implemented in the future Intel Xeon Phi processor and coprocessor known by the code name Knights Landing, and will also be supported by some future Xeon processors scheduled to be introduced after Knights Landing. Apart from Intel Xeon Phi coprocessors, it will therefore be very interesting to compare this approach with GPGPUs.

Acknowledgements

We would like to thank Wolfgang Lehner for the valuable discussions and suggestions for the paper and the SAP HANA team for the support in developing and testing our implementation.

6. REFERENCES

[1] Intel® 64 and ia-32 architectures optimization reference manual. Technical report, June 2013.

[2] Intel® architecture instruction set extensions programming reference. Technical report, July 2013. To Appear.

[3] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data - SIGMOD '06*, page 671, New York, New York, USA, June 2006. ACM Press.

[4] V. N. Anh and A. Moffat. Inverted Index Compression Using Word-Aligned Binary Codes. *Information Retrieval*, 8(1):151–166, Jan. 2005.

[5] V. N. Anh and A. Moffat. Index compression using 64-bit words. *Software–Practice & Experience*, 40(2):131–147, Feb. 2010.

[6] R. Dementiev, T. Willhalm, O. Bruggeman, P. Fay, P. Ungerer, A. Ott, P. Lu, J. Harris, P. Kerly, and P. Konsor. Intel© performance counter monitor - a better way to measure CPU utilization, 2013. `http://www.intel.com/software/pcm`.

[7] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 3(1-2):670–680, Sept. 2010.

[8] J. Feehrer, S. Jairath, P. Loewenstein, R. Sivaramakrishnan, D. Smentek, S. Turullols, and A. Vahidsafa. The oracle sparc t5 16-core processor scales to eight sockets. *IEEE Micro*, 33(2):48–57, 2013.

[9] G. Graefe and L. Shapiro. Data compression and database performance. In *[Proceedings] 1991 Symposium on Applied Computing*, pages 22–27. IEEE Comput. Soc. Press, 1991.

[10] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden. Performance tradeoffs in read-optimized databases. pages 487–498, Sept. 2006.

[11] A. L. Holloway, V. Raman, G. Swart, and D. J. DeWitt. How to barter bits for chronons. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data - SIGMOD '07*, page 389, New York, New York, USA, June 2007. ACM Press.

[12] Intel Corporation. ARK - your source for Intel© product information, 2013. `http://ark.intel.com/`.

[13] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd. Ibm's next-generation server processor. *IEEE Micro*, 30(2):7–15, 2010.

[14] L. Lamport. Multiple byte processing with full-word instructions. *Communications of the ACM*, 18(8):471–475, Aug. 1975.

[15] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. Sept. 2012.

[16] D. Lemire, L. Boytsovand, O. Kaser, M. Dionne, and L. Dionne. The FastPFOR C++ library : Fast integer compression, 2012. `https://github.com/lemire/FastPFOR`.

[17] C. Lemke, K.-U. Sattler, F. Faerber, and A. Zeier. Speeding Up Queries in Column Stores – A Case for Compression. pages 117–129, Aug. 2010.

[18] Y. Li and J. M. Patel. BitWeaving: Fast Scans for Main Memory Data Processing. In *SIGMOD*, 2013.

[19] P. O'Neil and D. Quass. Improved query performance with variant indexes. *ACM SIGMOD Record*, 26(2):38–49, June 1997.

[20] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-Time Query Processing. In *2008 IEEE 24th International Conference on Data Engineering*, pages 60–69. IEEE, Apr. 2008.

[21] M. A. Roth and S. J. Van Horn. Database compression. *ACM SIGMOD Record*, 22(3):31–39, Sept. 1993.

[22] B. Schlegel, R. Gemulla, and W. Lehner. Fast integer compression using SIMD instructions. In *DaMoN*, pages 34–40, New York, New York, USA, June 2010. ACM Press.

[23] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi. SIMD-based decoding of posting lists. In *CIKM*, page 317, New York, New York, USA, Oct. 2011. ACM Press.

[24] J. Wassenberg. Lossless asymmetric single instruction multiple data codec. *Software: Practice and Experience*, 42(9):1095–1106, Sept. 2012.

[25] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The implementation and performance of compressed databases. *ACM SIGMOD Record*, 29(3):55–67, Sept. 2000.

[26] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, Aug. 2009.

[27] M. Zukowski. *Balancing vectorized query execution with bandwidth-optimized storage*. PhD thesis, 2009.

[28] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. In *ICDE*, pages 59–59. IEEE, Apr. 2006.

APPENDIX

A. INFLUENCE OF THE COMPILER

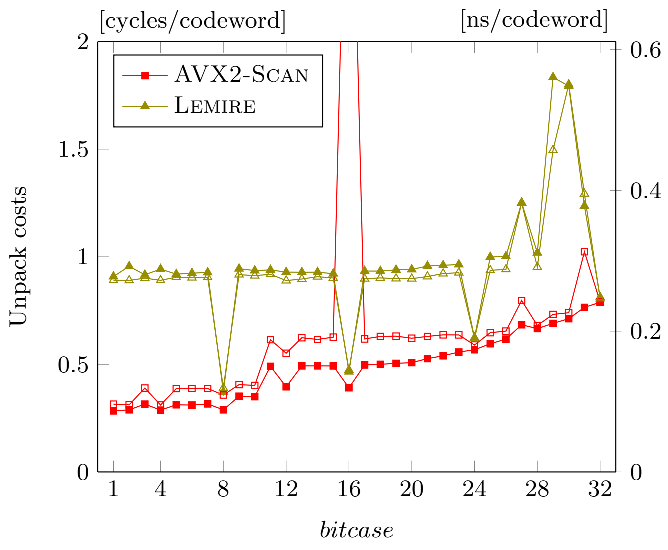

Figure 17: Pure unpacking costs of a full column-scan, compiled with ICC (solid markers) and with GCC (hollow markers).

Figure 17 shows an experiment justifying our choice to use different compilers for different routines. It shows the same experiment as Figure 12, but Lemire and AVX2-Scan are compiled with GCC (solid markers) and ICC (hollow markers). Lemire compiled with GCC is roughly 5% faster than compiled with ICC, while AVX2-Scan compiled with ICC is roughly 30% faster than compiled with GCC, where the latter even introduces a severe performance bug. Therefore we use GCC for Lemire and ICC for AVX2-Scan.