TECHNISCHE UNIVERSITÄT DRESDEN

Dep. of Computer Science Institute for System Architecture, Database Technology Group

Master Thesis

VECTORIZING RECOMPRESSION IN COLUMN-BASED IN-MEMORY DATABASE SYSTEMS

Cheng Chen

Matr.-Nr.: 3924687

Supervised by:

Prof. Dr.-Ing. Wolfgang Lehner

Submitted on 19. April 2015

Technische Universität Dresden/Fakultät Informatik

Master's Thesis Application

Name, First Name: Chen, Cheng

Course: Distributed Systems Engineering Student ID: 3 9 2 4 6 8 7

Subject:

Vectorizing recompression in column-based in memory database system

Objective :

SAP HANA is an in-Memory RDBMS. Data storage has two parts, main storage is for read operations, and the other called delta keeps differential updates for optimizing the write operations. Data in differential buffer will be merged into main storage in a certain time. In that process, values in main storage need to be first decompressed, merged with differential buffer and compressed again. This thesis would study the potential improvement and its implications of introducing SIMD instructions to accelerate the merge process.

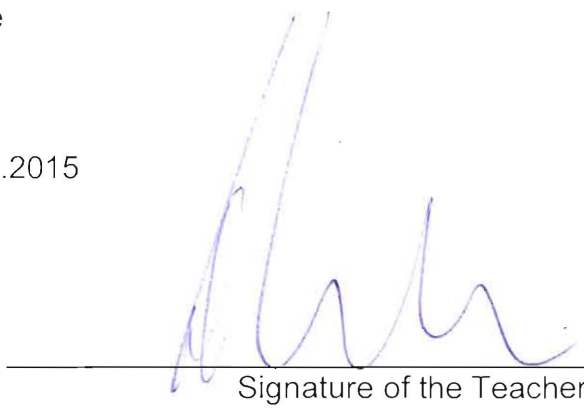
Supervisor: Prof. Dr.-Ing. Wolfgang Lehner

Teacher in Charge: Prof. Dr.-Ing. Wolfgang Lehner

Institute: System Architecture

Start date: 20.10.2014

The thesis must be submitted on: 19.04.2015



Signature of the Teacher in Charge

Verteiler: 1 x SCIS, 1x HSL, 1x Betreuer, 1x Student

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 19. April 2015

Contents

Abstract

With increasing size of memory in standard servers, In-Memory database systems are more popular nowadays. In-Memory database systems store most of or all relevant data in main memory. Many systems divide data into two parts: a main storage for read operations, and a differential buffer for optimizing the write operations. Data in the differential buffer is merged into main storage after a certain time. In that process, values in the main storage need to be first decompressed, merged with the differential buffer, and finally compressed again. This thesis studies the usage of SIMD instructions to accelerate the merge process.

As there are already several papers focused on accelerating the decompression process with SIMD instruction. This thesis will focus on compression. We also work on combining decompression and compression part together called Rollover.

Implementing SIMD compression, on one hand, will reduce the cost of memory access, since it reads more values at a time, and processing the several data in parallel, on the other hand, will make the algorithms more complex: compressed values are not always 8-bit aligned, they cross the byte boundary, so we cannot use single instructions to manipulate a single value.

The result of thesis presents that our fastest version SIMD compression routine is about 40% faster in the cases, with some cases being up to 2.4x as fast as their scalar counterpart. And for rollover, the fastest version is 6.6x as fast as the naive version.

1 Introduction

1.1 Background

In the past, database management systems were designed for optimizing performance on hardware with limited main memory and with slow disk I/O as the main bottleneck [KB14]. Therefore, in-memory database systems, which keep all relevant data in main memory, were designed to overcome the I/O bottleneck. Many systems divide data into two parts: a main storage for read operations, and a differential buffer for optimizing the write operations.
In order to keep all relevant data in memory, efficient compression algorithms must be used. However, write operations for in-memory database on compressed data would be costly, because it would require reorganizing the storage structure and re-compression after every update. Hence a differential buffer ensures the performance of write operations. Data in the differential buffer is kept uncompressed. A merge operation is executed in regular intervals, to move the changed data from differential buffer to main storage. As an example, the differential buffer in HANA is called "Delta". As Figure 1 shows, queries write to "Delta1" and read from both "Main1" and "Delta1". Merge process moves the data from "Main1" and "Delta1" to "Main2", and from that point of time new data start to write into "Delta2" until next merge process.

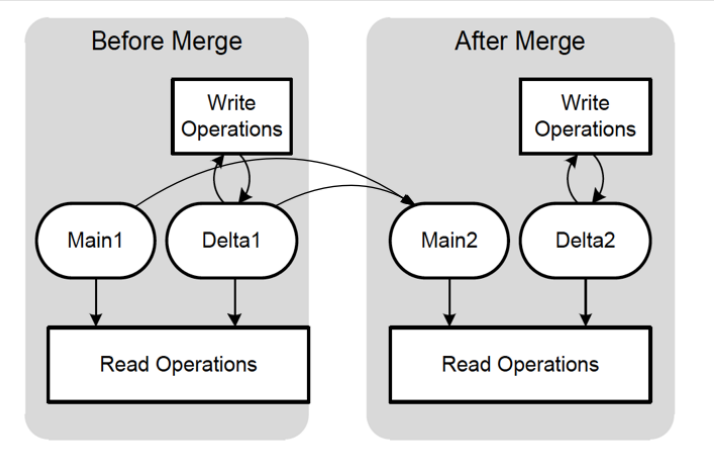


Figure 1: Merge Operation [KB14]

For column-oriented database, each column of values in the main storage is stored in a vector. These vectors use several compression techniques. For example, with dictionary compression, distinct values appearing in the column are stored in a sorted dictionary [KB14]. As it is shown in Figure 2, "Miller" is shown 3 times in the left column, "John" 2 times, "Millman" 2 times. In the right part of the Figure 2, distinct string values "Miller", "John", "Millman" are kept only once, and integer values(4-"Miller",1-"John",5-"Millerman") refer to the string value. In this thesis, we name these integers "Value ID".
With dictionary encoding, significant compression is achieved if the values in the column have low variance. The Value-ID array is compressed using minimum bits. For example, if the dictionary contains N values, the required number of bits is $log_2(N)$ [KB14]. Assuming the length of values varies from 1 to 32 bits, so we use term "bitcase" to refer to the length of a value. During the merge, Value-IDs are re-calculated. And after the merge operation, if new values were inserted, it will be necessary to remap the Value-ID array to a new dictionary.
In this process, the whole main storage will be rewritten, with huge amount of compression and decompression operations. For increasing the bitcase in main storage, one decompression and one compression will be needed, which needs to buffer huge amount of intermediate values in memory. A process called Rollover that combine compression and decompression consuming only $O(1)$ intermediate memory will be introduced in thesis. And we use SIMD instructions to accelerate these processes.

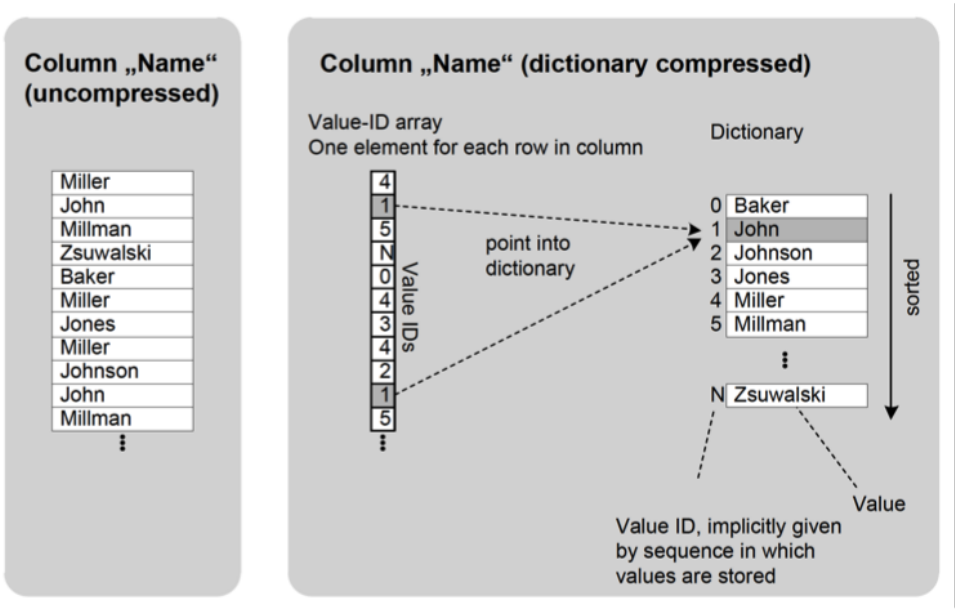


Figure 2: Dictionary Compression [KB14]

1.2 Merge Process in Existing Database

Some of the existing databases use similar mechanisms to optimize a mix of read/write operation.

1. SAP Sybase IQ PlexIM Merge

   SAP Sybase IQ is another column-based RDBMS. Sybase IQ acts more often as a data warehouse, while HANA is a real-time data platform. Sybase IQ also uses a similar mechanism as the SAP HANA Delta Storage, which is called PlexIM Store. Unlike in HANA, the main storage of IQ is in disk. PlexIM follows these rules [Flo14]:
   1) There is only one writable PlexIM store at any given time.
   2) Only, and all of the committed versions of rows in the PlexIM store will be merged into main store.
   3) Unified Row ID Space used in whole storage.

   To store the updated differential data, PlexIM Store keeps [Flo14].

   - A tail-insert array of tokens for each table column,
   - A single logical row deletion bitmap,
   - A tail-insert array of row metadata.

   When the merge operation is triggered, the uncommitted data is moved to a new PlexIM and the old PlexIM is written into the main store on disk.

2. C-Store

   C-Store is a read-optimized RDBMS from MIT. C-Store approaches the read-write operation dilemma by keeping one small “Writable Store” component, and a much larger “Read-Optimized Store” [SAB+05]. They are connected by tuple mover, as Figure 3 [SAB+05] shows.

   During the merge-out process (MOP), it will find all records and divide them into 2 groups. Those that were deleted before the merge-out process started are truly deleted. Those that were not deleted or were deleted after the process started are moved to a new Read-optimized Store (RS’). After RS’ is created, all data in WS and RS will be merged into RS’. The data in C-Store is

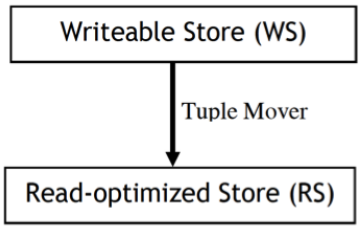


Figure 3: Architecture of C-Store [SAB+05]

compressed with 4 different schemes. The design of C-Store and the design of HANA have high similarity, except,

- In C-Store, RS and WS both use B-Tree to store the data, in HANA main storage is stored in vectors.
- In C-Store, the tuple mover is a process that keeps running in the background. After finding a certain (RS, WS) segment pair, it performs a merge-out process on it. In HANA, merge operation takes place on certain conditions, and it always rewrites the whole main storage.

1.3 SIMD instructions

For vectorizing the recompression in merge process, we use Intel provided SIMD instructions in this thesis.
Single instruction, multiple data (SIMD) is a class of parallel computers in Flynn's taxonomy. It describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously. For example, we have two vectors each with 4 elements inside, one instruction can offer 4 multiplications of each vector pair. As shown in Figure 4, the value with lowest address in first vector multiples the value in second with lowest address, and the rest values are also calculated in that order.

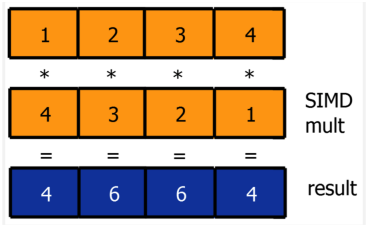


Figure 4: SIMD instruction example

Intel introduced the first MMX instructions in 1997. Later more instruction sets have been instroduced SSE with 128 bits wide vectors and AVX with 256 bits wide vectors, as listed in Table 1.
Intel provides intrinsic functions to use these SIMD instructions in the C programming language.

SSE4

SSE4 first appeared in the Penyrn architecture of Intel. It has 8 128-bit registers known as XMM0-7. With SSE4, it is possible to read 128 bits from memory, then process them simultaneously. In our case each value is 32 bits, so 4 values are processed in parallel.

AVX2

AVX2 further widens SIMD vectors (YMM0-YMM7) to 256 bits by extending 128-bit registers of SSE(XMM0-7), and it provides eight more 256 registers (YMM8-YMM15). So definitely it has the

Date	SIMD set	size of registers(x32 bits)	new features
1996	Intel MMX	2	integer
1999	Intel SSE	4	float
2001	Intel SSE2	4	float and integer
2004	Intel SSE3	4	some improvements
2006	Intel SSE4.2	4	8-bits operations
2008	Intel AVX	8	float
2013	Intel AVX2	8	float and integer
2015	Intel AVX-512	16	float and integer
unknow	Intel AVX-512BW	16	+8 and 16-bits operations

Table 1: Historic evolution of SIMD instruction sets

ability to load more data with one operation. But 256-bit registers (YMM0-YMM15) are divided into two 128-bit lanes: a low lane and a high lane. Except a few instructions, AVX instructions operate within a lane: the result in a certain lane is calculated using data only from the same lane. Cross-lane instructions such as the cross-lane shuffle "$\_mm256\_permutevar8x32\_epi32$" are offered to exchange the data between 2 lanes. However, it takes more CPU cycles than in-lane operations, as it is shown in Table 2.

Instruction	Architecture	Type	CPU cycles
$\_mm256\_permutevar8x32\_epi32$	Haswell	cross-lane	3
$\_mm256\_shuffle\_epi32$	Haswell	in-lane	1

Table 2: cross-lane vs in-lane shuffle instructions

Non-Temporal Write

There is one final feature we take advantage of Intel architecture. Because the CPU core is not directly connected to the main memory, each cache line is mapped to memory. As shown in left part of Figure 5, when data is written to memory via cache, the corresponding cache line in memory first loaded in cache, then the CPU will write data to this cache line. Only then is the data transferred from cache to memory. If the data is not going to be used within a short period of time, copying memory to cache is wasteful. The Intel Architecture offers an alternative: non-temporal write instructions. Non-temporal writes are a mechanism to bypass the cache and write the data directly to memory, thus saved reading the cache-line, as the right part of Figure 5 shows.

1.4 Thesis Statement and Contributions

1.4.1 Thesis Statement

In this thesis we focus on studying the compression process and the rollover process which combines decompression process and compression process together. It proves that SIMD instructions can accelerate both processes.

1.4.2 Contributions

This thesis proposes vectorized approaches of compression and rollover:

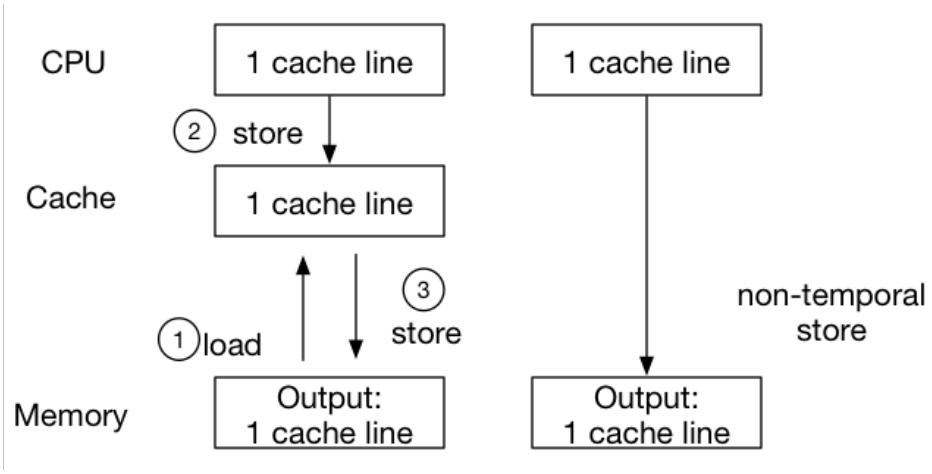


Figure 5: Normal store instructions VS Non-Temporal Store instructions

- Compression: During the merge process, values are first decompressed into 32 bits intermediate values, and then compression is executed to transfer values into target bitcase. We introduce faster SIMD algorithms to compress values using SSE4 and AVX2 instructions.

- Rollover: In some scenarios, values are not processed after decompression. Only a compression is executed direct after that. Generating a huge amount of intermediate values is considerable waste in this case. In this paper we introduce new algorithms to combine decompression and compression together.

The rest of thesis is organized as follows: We first discuss related work in Section 2. It covers different compression schemes, the standard bitpacking algorithm that we use in this thesis, other bitpacking algorithms and some other compression schemes. And we will also talk about scan and decompression process with SIMD instructions. In Section 3, we give details of our compression algorithm with SIMD instructions. In Section 4, we explain the algorithm of Rollover, which combines compression and decompression together. Section 5 presents and discusses performance evaluation of both compression and rollover. Finally, we draw conclusions from the thesis and provide an outlook in the last section.

2 Related Work

In this section, we relate our thesis to existing work. We first explain the bitpacking format, and SIMD decompression and SIMD scan with this bitpacking format. Then we review some work that studied different compression schemes.

2.1 Standard Bitpacking

Bitpacking is a format used for compressing Value-ID vectors in main storage. Different compression scheme can be used to achieve that purpose. The idea is to use as few bits as possible for Value-IDs. As Figure 6 shown, before compresssion, each value possesses 32 bits. 32 bits can at most represent 2^{32} values. If we only have values from 1 to n, $log_2(n)$ bits(we use b to represent $log_2(n)$ in the figure) are enough. Therefore a compression process transforms data from 32 bits to $log_2(n)$ bits. The bit packing process in the figure is to convert data from lower part to the higher part.
The compression speed is as important as the compression rate. Therefore, vectorizing the merge process is an interesting topic. There are previous works that focused on SIMD scan and SIMD decompression. Two papers [WBP+09] and [WOMF13] of Willhalm et al. discussed how to use VPUs on standard superscalar processors accelerate the full table scan. [WBP+09] uses Intel Streaming SIMD Extensions

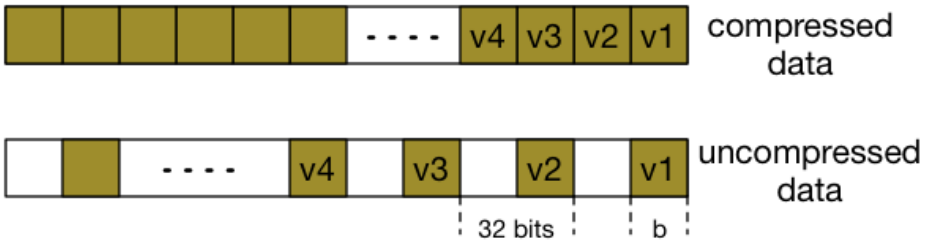


Figure 6: HANA compression scheme

(Intel SSE) with 128-bit registers and [WOMF13] uses the Intel AVX2 instruction set to accelerate vectorized scans with complex predicates. SIMD Algorithm for decompression has also be discussed in [WOMF13].

As it is shown in the first vector of Figure 7, Value-IDs are not 8 bits aligned. The processor can only process data 8, 16, 32, 64 bits aligned, therefore decompression is necessary.

Decompression is already studied using SIMD with SSE, AVX2 instructions. For SSE version, 128 bits can be decompressed at one time. Data will be processed in the following order in Figure 7.

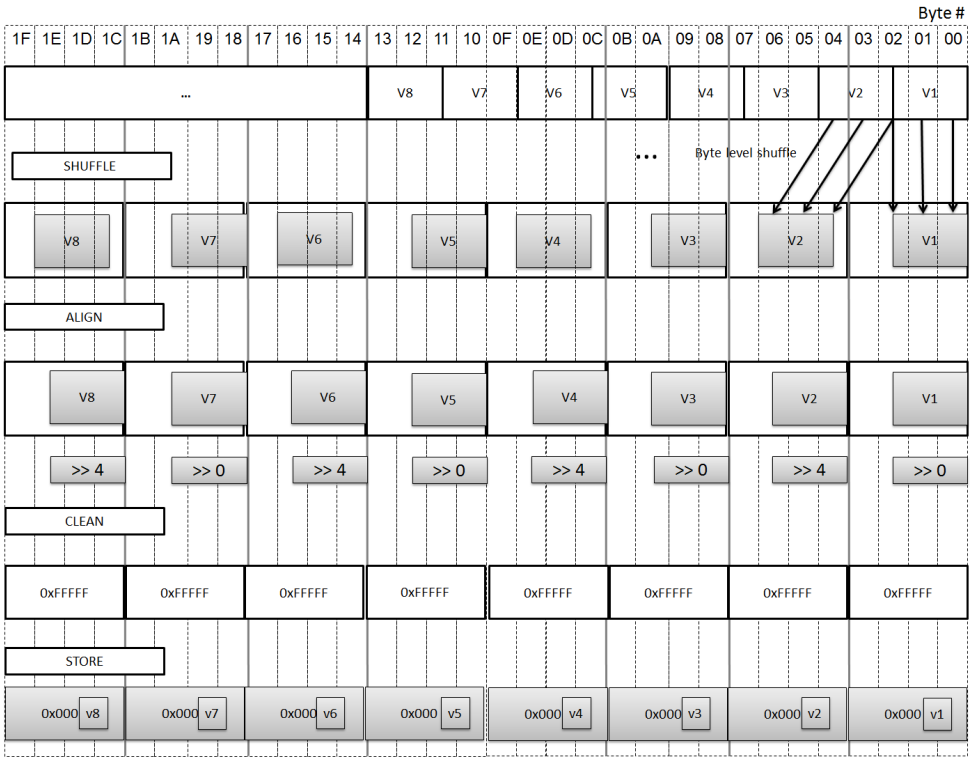


Figure 7: Algorithm of Compression [WOMF13]

In Figure 7, each value is 20 bits long, and the decompressed value will be 32 bits long. The first step is shuffle. In this step, shuffle instructions are used to put the right Value-ID into the corresponding 32 bits. After that, in the second 32 bits, the first 4 bits are from first value, then 20 bits of the second value follow, after that the rest 8 bits are from the third value. In the next step, shift instructions will be used to make the data 32 bits aligned. The last step is to set the non-relevant bits to 0 using mask. After that we can extract the values from each 32bits and store them back into memory or process them.

2.2 Bitpacking Variants

There are other compression schemes that may also be relevant to main index compression. One of them, Lemire's FastPFor provides a different scheme for compression. Lemire's algorithm stores values as Figure 8, if the vector is 128 bits long and each uncompressed value contains 32 bits. The first vector contains (int1,int2,int3,int4), and the second (int5,int6,int7,int8). During the compression process, we put the int5 directly next to int1, and int6 next to int2. So the interval of adjacent values are 4 inside 32 bits. If the last value cannot be completely stored inside these 32 bits, the rest of it will be stored in next 128 bits.

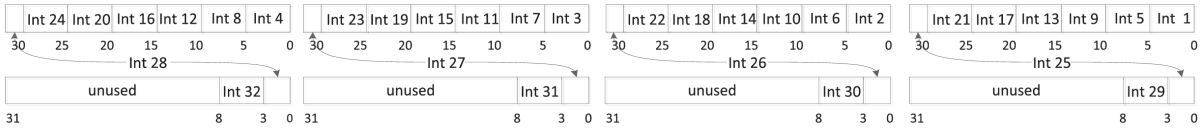


Figure 8: Lemire's Compression Scheme [LB13]

In that order, the compression process is simplified. It keeps the distance between values the same as it is loaded. Value-IDs don't need to be shuffled and moved horizontally in a SIMD vector. The SIMD registers just need to "shift" and "or" operations to store the data into a compressed order as Figure 9 shows. The calculation on each value is just the same as the generic version, only vectorized version is processing 4 values in parallel at a time.

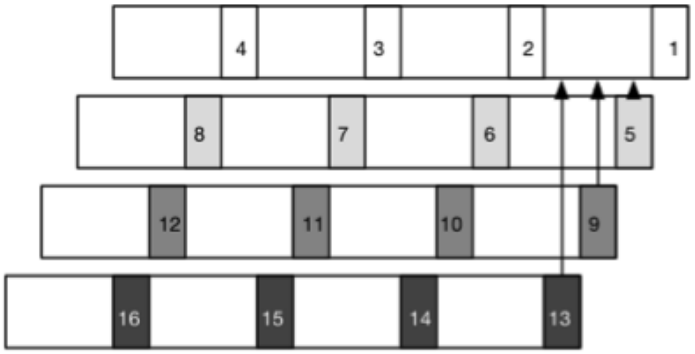


Figure 9: Lemire's Compression Method [LB13]

Therefore, it will certainly achieve a better compression speed, ideally by factor 4. And for decompression, the only additional operation is to re-calculate the order of Value-ID, which is possibly not losing much decompression speed.
The standard bitpacking is also referred as horizontal bitpacking, Lemire's FastPFor is sometimes referred as vertical bitpacking. Detailed comparison of Horizontal Bitpacking, Vertical Bitpacking or Aligned Vertical Bitpacking is discussed in a paper by Faust et al. [FGB+14].

2.3 Other Compression Schemes

There are other compression schemes listed below, which focus on bit level, and are easy for compression and decompression.

Run Length Encoding Run Length Encoding replaces sequences of the same value with a single instance of the value and its start position. This variant of run length encoding was chosen, as it speeds up access compared to storing the number of occurrences with each value [KB14], as Figure 10 shows.

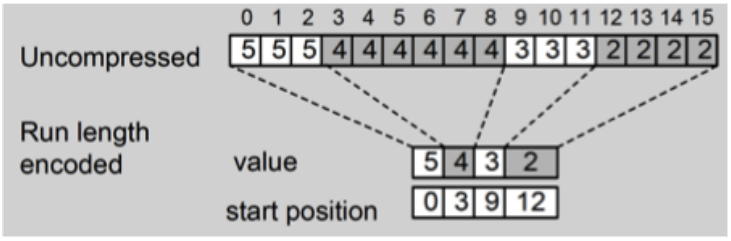


Figure 10: Run Length Encoding [KB14]

Sparse Encoding Sparse encoding removes the value V that appears most often. A bit vector indicates at which positions V was removed from the original array [KB14], as Figure 11 shows.

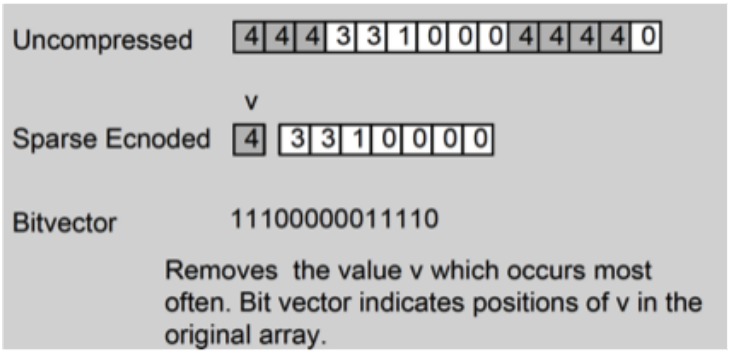


Figure 11: Sparse Encoding [KB14]

3 Compression

We already discussed the merge process and the decompression process from previous work in Section 1.1 and Section 2.1. In this section we provide details of our SIMD compression algorithm.
A naive algorithm could work as Figure 12 and Algorithm 1: we load 32 bits once at a time, and shift value next to the most significant bit of the previous one. Finally we use a logic "or" operation to combine the two values together, which takes 1 "load", 1 "shift" and 1 "or" for each value, as Figure 13 depicted.

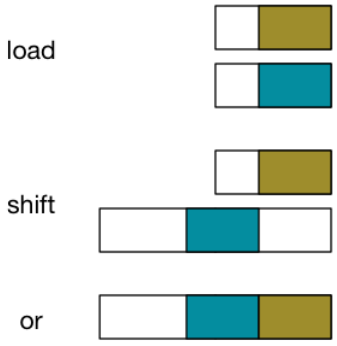


Figure 12: Naive Compression of one value

The basic idea of SIMD compression remains the same. For vectorizing the process, the challenge is that there is no operation to gather data in arbitrary length from each 32 bits segments into a vector, which is the opposite process of first step in decompression (SHUFFLE) in Figure 7.

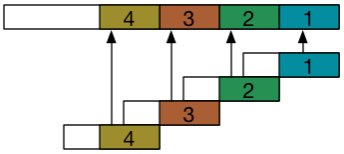


Figure 13: Naive Compression of a sequence of values

Algorithm 1 Generic Algorithm Bitcase b

```
set k to 0
for i from 0 to max_index/(32 * chunk_size) do
    for j from 0 to chunk_size do
        load r_j from input + j
        shift left r_j by b
        or r_j, r
    end for
    store r
end for
```

3.1 Algorithm

The packing algorithm varies from bit case to bit case. We group the algorithms into 5 classes. We explain the algorithm based on Intel SSE instrutions which have 128 bits each vector. For AVX2, the vector contains two lanes each 128 bits. Together is 256 bits long, which can process 8 values at a time, but cross-lane operation takes lot more time. There are small differences between the AVX version of the algorithm and the SSE version. We will talk about the difference after illustrating the algorithm. For AVX, we have an additional algorithm that can be quite efficient.

Bitcase 1 In this bitcase, there is only one bit in each 32 bits. The purpose is to extract the value from the least significant bit. To do that we shift the least significant bits to the most significant bits and take advantage of special instructions to gather all most significant bits in vector. Described in Figure 14, we process 4 vectors at a time.

Algorithm 2 Bit packing bitcase 1

```
set k to 0
for i from 0 to max_index/(128 * 4) do
    parallel_load v_1 from input[i * 128 + 128 * 0]
    parallel_load v_2 from input[i * 128 + 128 * 1]
    parallel_load v_3 from input[i * 128 + 128 * 2]
    parallel_load v_4 from input[i * 128 + 128 * 3]
    parallel_shift v_1 by 7 bits
    parallel_shift v_2 by 7+8 bits
    parallel_shift v_2 by 7+8*2 bits
    parallel_shift v_2 by 7+8*3 bits
    parallel_or v_1 and v_2 -> v_12
    parallel_or v_12 and v_3 -> v_123
    parallel_or v_123 and v_4 -> v_1234
    shuffle v using shuffle_mask(m_15, ..., m_0)
    extract MSB of each byte with movemask into r
    store r
end for
```

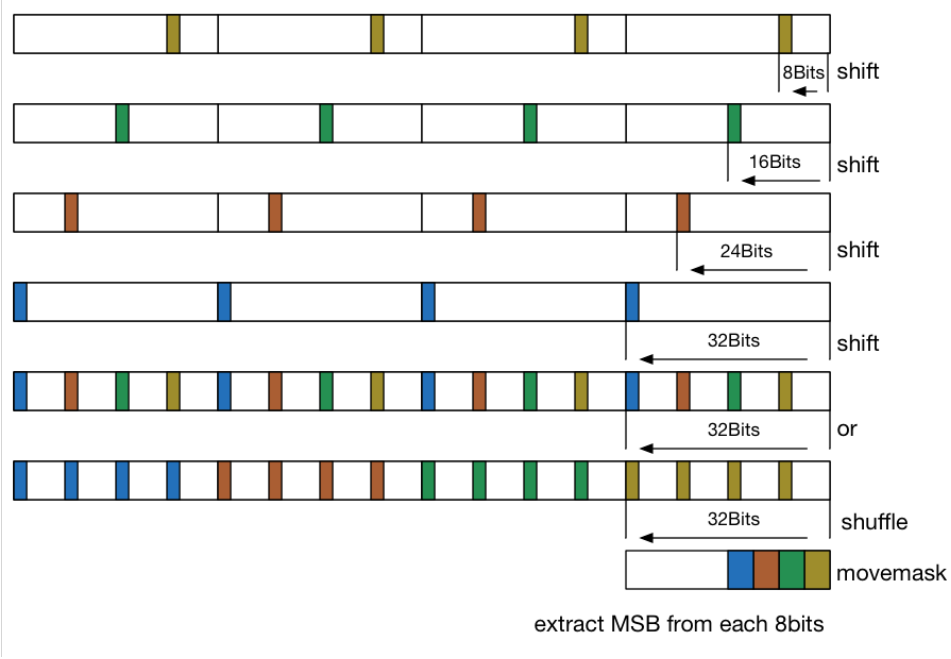


Figure 14: Bitcase 1

First, we load 4 vectors (Algorithm 2, line 3-6), and shift each to position 7, position 15, position 23, position 31 (line 7-10). Then we use "or" operations to combine vector together (line 11-13). After that, each byte contains one bit value, which is the most significant bit of the byte. We need one more shuffle (line 14), to exchange the order (start from 0) from [15,11,7,3,14,10,6,2,13,9,5,1,12,8,4,0] into the right order [15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0]. Finally, we benefit from special instruction to gather data from the most significant bit of each byte (line 15). For each 16 values, we need 4 loads, 4 shifts, 3 ors, 1 shuffle and a movemask operation.

For AVX, we can load 256 bits at a time. With the movemask operation we get exactly 32 bits at a time. The algorithm itself remains the same.

Bitcase multiples of 8 For these special bit cases, ValueIDs align with byte boundary. We can certainly benefit from special Intel instruction "shuffle", which re-arranges the order of bytes of a SIMD vector.

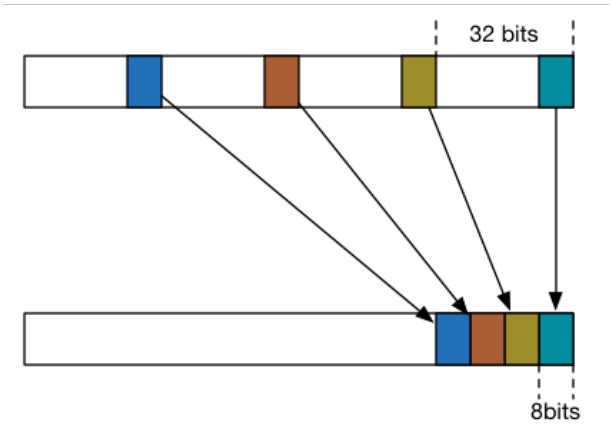


Figure 15: Algorithm for Bitcase multiples of 8

Algorithm 3 Bit packing bitcase multiples of 8

1: **for** i from 0 to $max\_index/128$ **do**
2: parallel_load v from input[$i * 128$]
3: shuffle v using shuffle_mask($m_{15}, ..., m_0$)
4: extract 32 bits into r
5: store r
6: **end for**

The simplest example is bit case 8, shown in Figure 15 and Algorithm 3. We load one vector at a time load (line 3), and one simple shuffle follows (line 4). For using 128 bits instructions, we get 32 bits as result. However, extracting 32 bits from a vector register is costly and storing 32 bits once at a time is also less efficent. A better way is to buffer four results, combine them together, and write them once to the memory.

As for AVX, after the shuffle, values are continuous in the lower part of each lane. Then we need a cross lane permute to exchange between the two lanes.

Bitcase 9-31 except 16 and 24 In bit cases of this region, values cross byte boundary, there is no special instruction that we can use. And for these, we design two different algorithms. We design Algorithm 4. We merge two values in each half of the vector (line 5-7) and then combine the values in the lower half and the higher half of the vectors (line 8-10). Figure 16 shows bitcase 17 as an example. After the first and second phase, the values are continuous. We extract the data in first 64 bits (line 12) and store and save the remaining 4 bits in another vector for the next iteration (line 13). Because Intel doesn't support shift of whole 128-bit registers, we have to shift packed 64 bits, 4 bits will lost (line 11). Thus the remaining 4 bits need additional instructions to be extracted from original vector.

Algorithm 4 Compression for bitcase 9-31 except 16 and 24 without carry-over

1: set k to 0
2: **for** i from 0 to $max\_index/(128 * b)$ **do**
3: **for** j from unrolling b times **do**
4: parallel_load v from input[$i * 128 + j$]
5: mask $v[127:96, 95:64, 63:32, 31:0]$ –> $v_{02}[0, ..., 0, 95:64, 0, ..., 0, 31:0]$
6: shuffle $v[127:96, 95:64, 63:32, 31:0]$ –> $v_{13}[0, ..., 0, 127:96, 0, ..., 0, 63:32]$
7: shift left v_{13} by b –> v_{13}
8: v_{02} parallel_or v_{13} –> v_{new}
9: mask $v_{new}[127:64, 63:0]$ –> $v_{new02}[0, ..., 0, 31:0]$
10: shuffle $v_{new}[127:96, 95:64, 63:32, 31:0]$ –> $v_{new13}[0, ..., 0, 95:64]$
11: shift left v_{new13} by $2 * b$ –> v_{new13}
12: v_{new02} parallel_or v_{new13} –> r
13: concatenation r and p $r|p$
14: extract and store lowest $(b * 4div32) * 32$ bits in r
15: save $b * 4mod32$ bits highest of r in p
16: **end for**
17: **end for**

There is another special algorithm we can use to compress values. Because for the previous algorithm, the main problem is that Intel only provides two ways to shift the register: one is to shift the whole register by bytes, the other is to shift each 32/64-bit word by bits. So there is no instruction to shift register across the 64 bits boundaries. Therefore keeping the remainders needs complex operations.

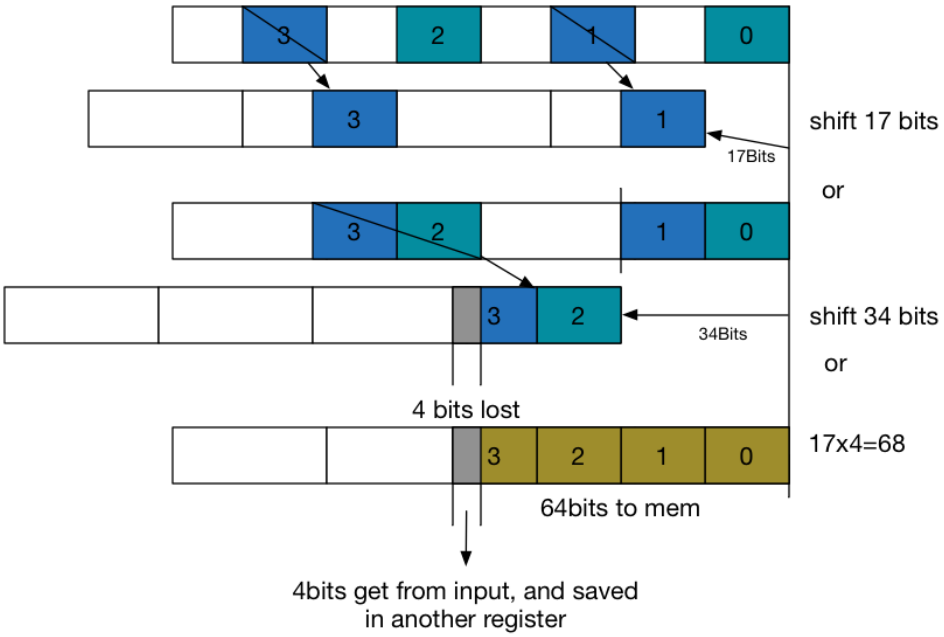


Figure 16: Compression for bitcase 9-31 except 16 and 24 without carry-over

Here we introduce a new algorithm, which benefits from a new feature in AVX2: shift each 32/64 bits with a different offset.

We illustrate the approach by explaining it just only for values 0 and 1. We separate values across the boundary into two different parts. Figure 17 shows the operation on the first two values of algorithm with in bitcase 17. As we can see in step 2a, we shift the second value to the left by 17 bits, in order to leave space for the first value. In that step, we lost 2 bits at the end of the shift. We shuffle the second 32 bits to the first, and buffer them in another register. In step 2b, we shift the second value right by 15 bits, so there will be only 2 bits left in the second 32 bits, which are the bits lost in step 2a. At last by "or" these two register produced by 3a and 2b, we have achieved the compression of the first and the second value. We can use the same method to accomplish the compression algorithm.

The full algorithm of first iteration is shown in Figure 18. We also illustrate the algorithm step by step in Algorithm 5. Line 4 changes data from step 1 to step 2a in the figure. Line 5 follows to change data from 1 to 2b, and we need one more shuffle to deal with the position of values in line 8. In line 4, 5 we have 2 shift offset vectors and line 6-8 we need 3 shuffle masks. For these mask, we need to pre-calculate them. In line 11, we concatenate the v_5 and $previous\_left\_value$.

The most tricky part is still dealing with the bits left from last iteration. As we illustrate in Figure 19, for two different iterations, we need to calculate different shift offsets for each of the packed 32 bits, and we also need to calculate different shuffle masks for it. In Figure 19, the first shift offset vector starts at 0. The next iteration combines with value left from the last iteration, so the shift offset starts from 8, then 16, and then 24. For using 24 bits from the last iteration, we only need 7 values of current loaded vector. So we either save the last bits 7th value and 8th value or abandon them. For bitcase 17, at 17th iterations, the rest bits will be 0.

For dealing with the rest, there are actually two ways: Saving the bits for the next iteration, or abandoning the rest to do an unaligned load in the next iteration.

Bitcase less than 8 For these bitcase, Algorithm 4 for bitcase 9-31 also works. But one iteration only produces less than 32 bits, so we need to concatenate two or more iterations together to fill 32 bits

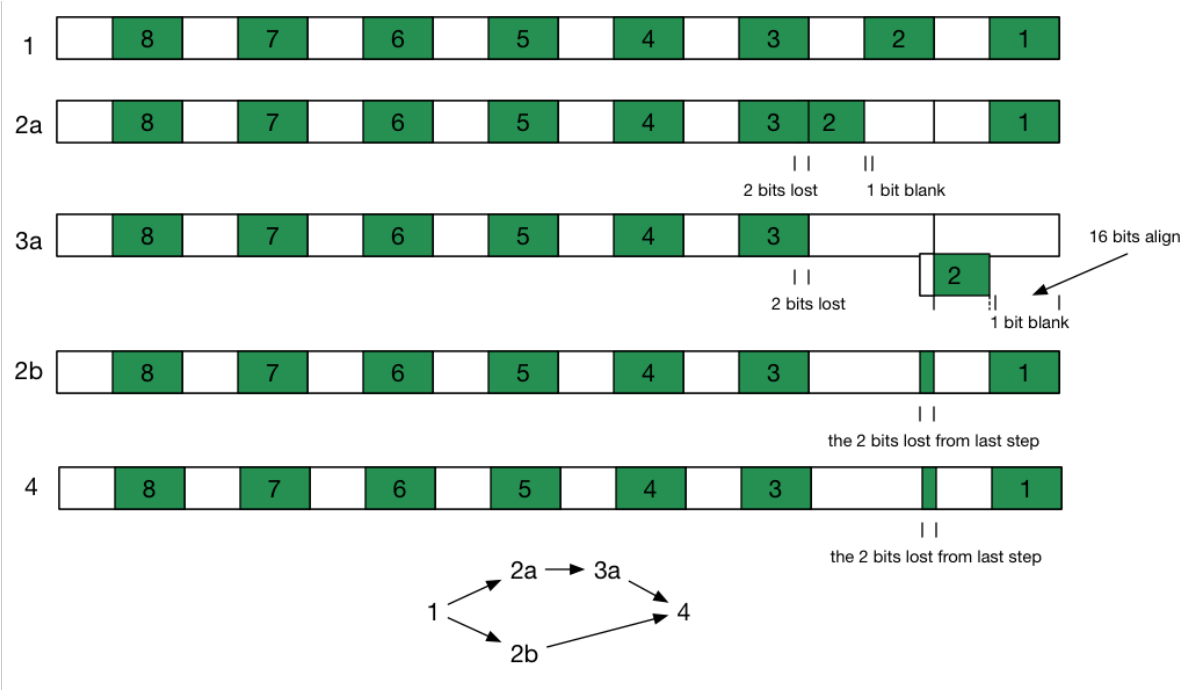


Figure 17: Compression for bitcase 9-31 except 16 and 24 with carry-over

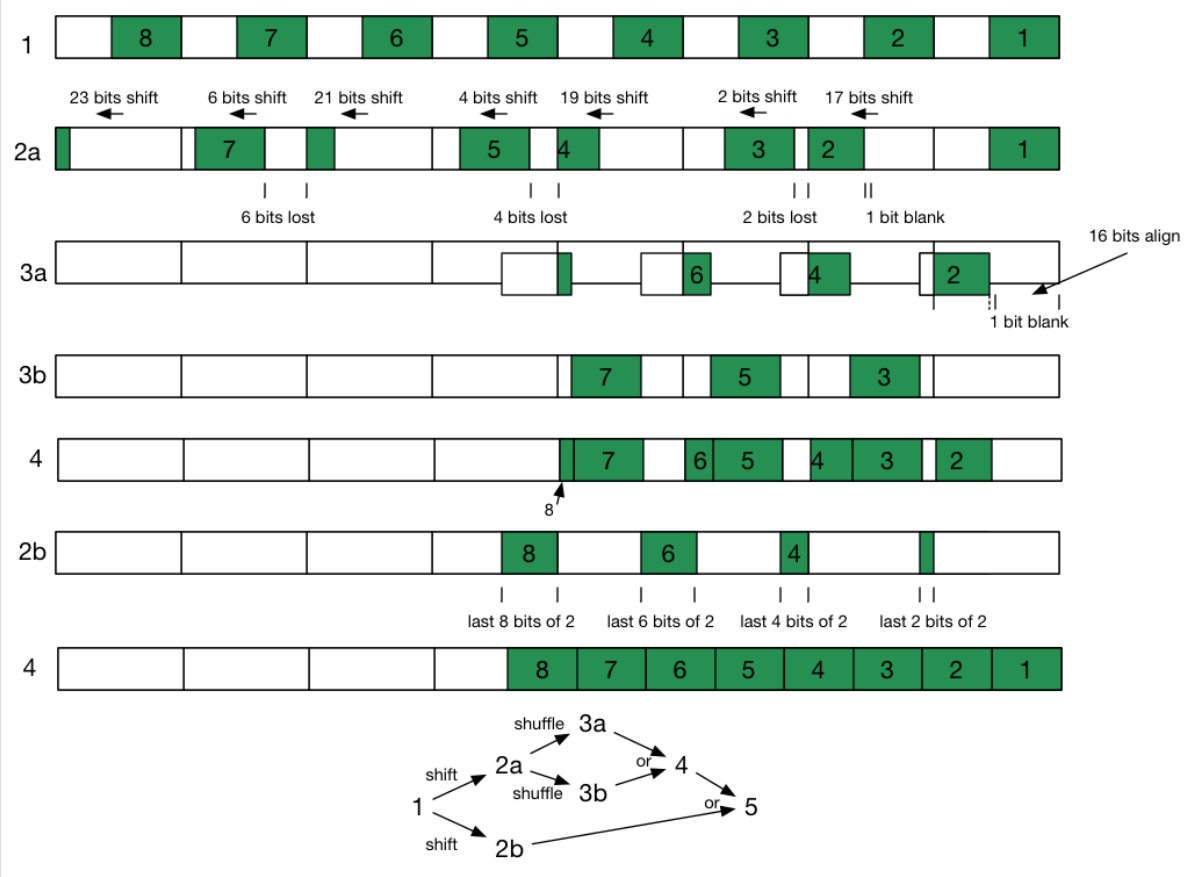


Figure 18: full process of new compression algorithm

Algorithm 5 Compression for bitcase 9-31 except 16 and 24 with carry-over

set k to 0
for i from 1 to $max\_index/(b * 8)$ **do**
  for j from 1 to b **do**
    parallel_load v from input
    v_{2a} <– parallel_shift v by $sftv(a_1, a_2, a_3, a_4, a_5, ..., a_8)$
    v_{2b} <– parallel_shift v by $sftv(b_1, b_2, b_3, b_4, b_5, ..., b_8)$
    v_{3a} <– v_{2a} shuffle by mask $(m_0, ..., m_8)$
    v_{3b} <– v_{2a} shuffle by mask $(n_0, ..., n_8)$
    v_{2b} <– v_{2b} shuffle by mask $(p_0, ..., p_8)$
    v_4 <– parallel_or v_{3a}, v_{3b}
    v_5 <– parallel_or v_4, v_{2b}
    v_5 <– parallel_or v_5, $previous\_left\_value$
    store 128 bits of v_5
    $previous\_left\_value$ <– bits left from v_5
  end for
end for

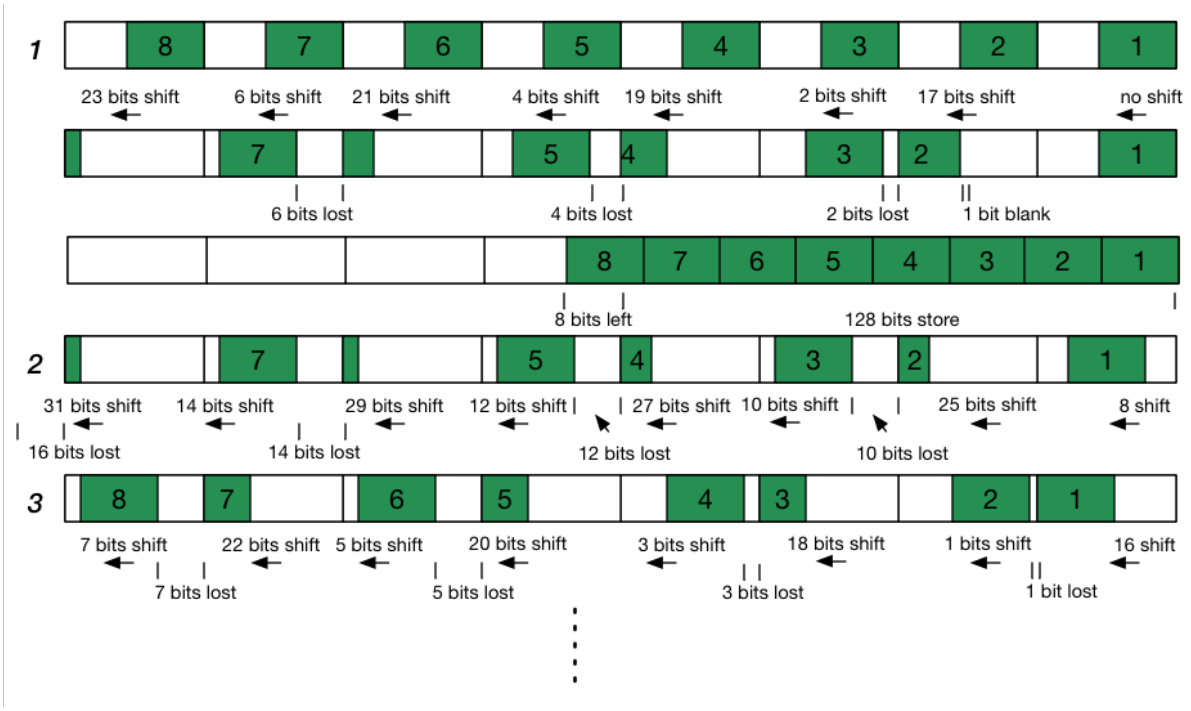


Figure 19: Shift offset vector and dealing with rest bits

for storing. By packing 4 vectors in parallel at a time, we can save half of the calculations, as it is shown in Algorithm 6. Figure 20 provides bitcase 2 as an example, values in the same color should be continuous. In line 4-6, we leave 3 values blank space in between, which is saved for value in the same vector. Furthermore we abandon the unused bits. In the next iteration, we just do an unaligned load and process again. This saves a mask operation and concatenating the unused bits in next iteration.

For these bitcases, Algorithm has no difference between AVX and SSE.

Algorithm 6 Compression bitcase less than 8

1: set k to 0
2: **for** i from 0 to $max\_index/(128 * b * 4)$ **do**
3: parallel_load v_1,v_2,v_3,v_4 from input
4: parallel_shift v_2 by $1 * (b * 4)$
5: parallel_shift v_3 by $2 * (b * 4)$
6: parallel_shift v_4 by $3 * (b * 4)$
7: parallel_or v_1,v_2,v_3,v_4 into v_{new}
8: using shuffle split v_{new} into 4 vectors, each with 32 bits, v_{new1},v_{new2},v_{new3},v_{new4}
9: parallel_shift v_{new2} by $1 * b$
10: parallel_shift v_{new3} by $2 * b$
11: parallel_shift v_{new4} by $3 * b$
12: parallel_or v_{new1},v_{new2},v_{new3},v_{new4} into r
13: extract and store lowest 32 bits in r
14: **end for**

Figure 20 provides bitcase 2 as an example.

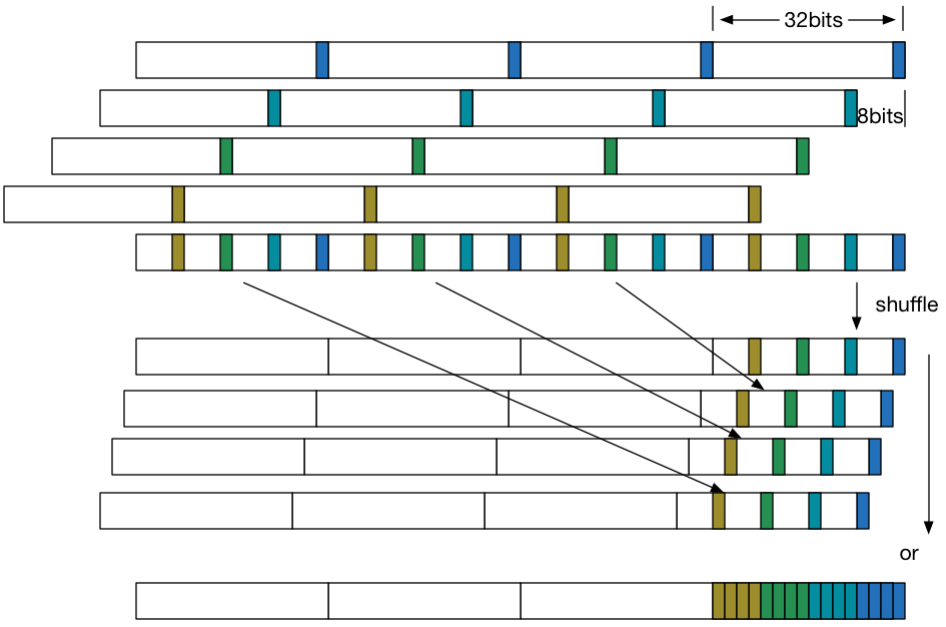


Figure 20: Less than bitcase 8

Bitcase 32 If the values in the vector are already 32 bits wide, obviously compresssion is not necessary, only a simple memcpy will finish the job.

As we illustrated in this section, we provide a group of algorithms to compress Value-ID vectors with different bitcases. We have a special algorithm which benefits from new AVX2 instructions.

4 Rollover

4.1 Concept

In Section 1.1, we have also talked about decompression. As Section 3 illustrated, there are five different algorithms to compress different bitcases. Lots of different jobs can be done with them. There is one operation that executes on the whole main storage combines decompression and compression processes to convert a huge amount of values from one bitcase to another. In this thesis, we name it Rollover.
For each column in the main storage we have two data structures, a vector for Value-IDs, and a dictionary for mapping Value-IDs to Values. Each Value-ID is mapped to a value, as it is shown in Section 1.1 Figure 2. In the merge process, after inserting or removing a certain amount of values, the number of bits required for representing Value-IDs may change. Each Value-ID of this vector in main storage should increase or decrease by Δb. In a general merge process, for example as Figure 21 shows, we have 3 values in main storage and 3 values in differential buffer. Each have Value-IDs from 0 to 2. After merging the values into a new dictionary, the Value-IDs have changed and have to be remapped to their new values. In dict2, the Value-ID of b is 0, and the Value-ID of f is 2, but after the merge the Value-ID of b changed to 1 and f to 5 in dict3. It means that the dictionary is remapped. If all Value-IDs changed, we cannot use rollover. But there are two scenarios listed below that are suitable for Rollover.

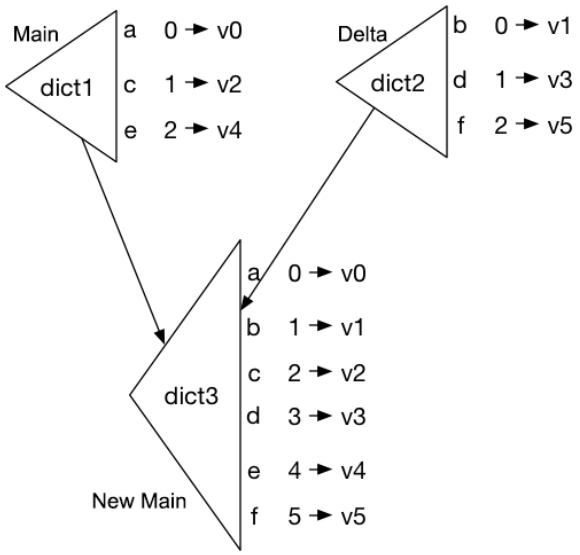


Figure 21: Merge Process in Detail

Last value of main storage < first value in differential buffer For the first scenario, all values in the differential buffer are bigger than the values in the main storage. In application level, it happens when values of this column automatic increase. The content of ValueIDs in main storage remain unchanged, only the bitcase does. So we just need to extend each value in the main storage and append all values in the differential buffer at the end of it after compression. For extending each value in the main storage, we use rollover. As Figure 22 shown, v2 is smaller than v3. Therefore the Value-IDs of a, b, c are unchanged, so we can simply use Rollover to extend the bits from 2 to 3, append and remap the Value-IDs differential buffer.

Main storage is continuous inserted into Differential Buffer In the second scenario, Values in main storage are all going to a continuous block in the new dictionary. Then the Value-IDs actually all change, but only by the same offset. We use rollover for this part of data and add the a constant. As Figure 23 shows, c is greater than b and e is smaller than f. So the Value-ID of dict1 0-2 become 2-4, we need a rollover process to extend the bitcase from 2 to 3, and we add 2 for the offset.

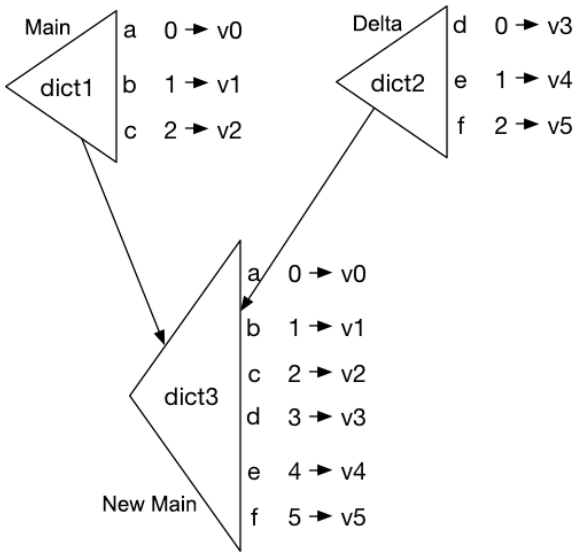


Figure 22: Rollover Scenario 1

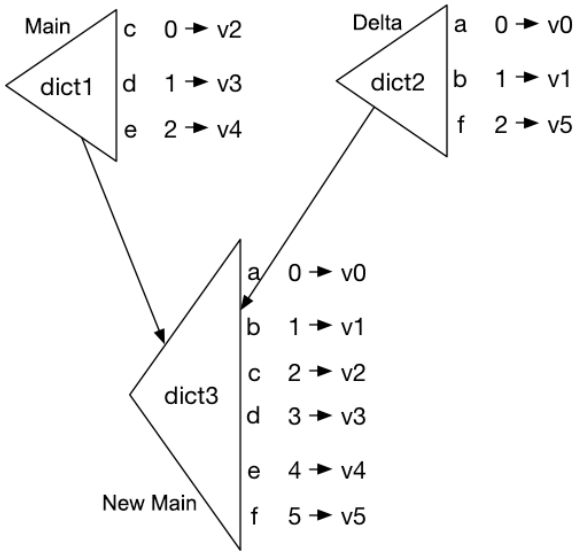


Figure 23: Rollover Scenario 2

As Figure 24 shows, a general method for both cases is to execute the decompression method once to convert the values into 32 bits intermediate values and the execute the compression method once to the target bitcase.

Algorithm 7 Naive Rollover

1: buf <– $decompression(ValueID\_in\_main)$
2: buf <– $offset(buf)$
3: $dest$ <– $compression(buf)$

For this process, a buffer is needed in memory and it needs more memory access operations. For saving the memory space and memory access time, a new algorithm is needed.

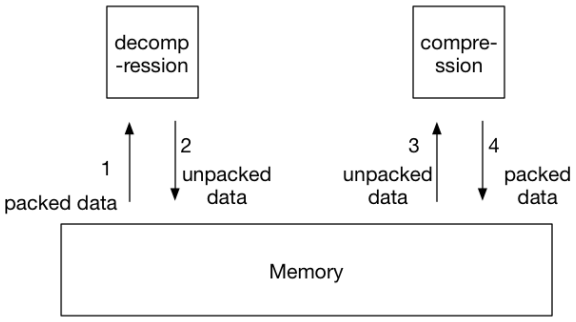


Figure 24: Naive Rollover

4.2 Algorithm

Since the general method is not efficient, we introduce 4 new algorithms in this section and compare them with each other.

Blockwise Rollover The first obvious algorithm is to process Value-IDs blockwise. The intermediate values of each block can fit in level 1 cache. This is shown in Figure 25. We decompress one block of values, and then write them into cache. After that, compression transforms the data in cache from 32 bits to the target bitcase and then writes it back to memory use non-temporal stores.

Algorithm 8 Blockwise Rollover

1: **for** i from 0 to $max\_index/block\_size$ **do**
2: buf <– $decompression(ValueID\_in\_main)$
3: buf <– $offset(buf)$
4: $dest$ <– $compression(buf)$
5: **end for**

For example, if the level 1 data cache is 32KB, and we use quarter of the cache to buffer the intermediate values. Each integer is 4 bytes, so we can buffer 2048 values at a time.

SIMD Rollover For SIMD rollover, we still choose 32bits value as intermediate value. Values are read into a SIMD register, then unfold into 32bits values in several registers, and again packed into the new bit case without writing intermediate values into memory, as shown in Figure 26. With this algorithm, data does not write back to cache or memory. For accessing register, we need only 1 CPU cycle. Accessing level 1 cache needs 1 ns (around 2 CPU cycles), and accessing memory takes around 100 ns. Therefore, it shorten the load and store time.

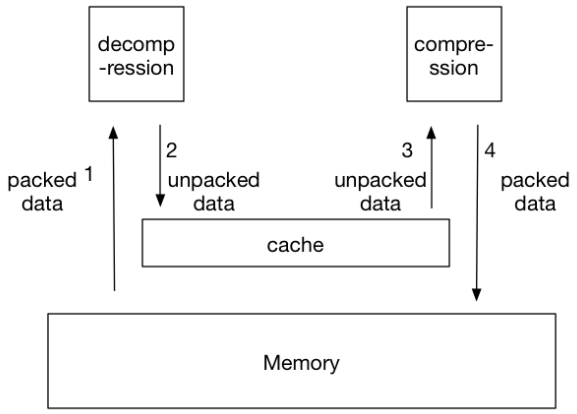


Figure 25: Blockwise Rollover

Algorithm 9 SIMD Rollover

1: $simd\_register$ <– $decompression(ValueID\_in\_main)$
2: buf <– $offset(simd\_register)$
3: $dest$ <– $compression(buf)$

An ideal way to do SIMD rollover is to use as few bits as possible for the intermediate values. The reason we need intermediate values is that we cannot process values that are not byte aligned. For smaller bitcases, we actually don't need to transform into 32 bits. The rules of the size of intermediate values are as follows: In the interval $[1, 8]$, we use 1 byte, in the interval $[9, 16]$, we use 2 bytes, for $[17, 24]$ we use 3 bytes, and for $[25, 32]$ we use 4 bytes.

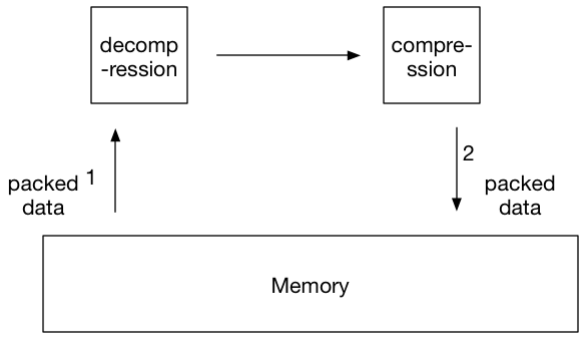


Figure 26: SIMD Rollover

BMI2 Instructions The three Rollover algorithms above still need an intermediate buffer to buffer 32-bit intermediate values. Difference is in general version we use memory, in the blockwise version we use cache, and in SIMD rollover we use register with these 32-bits values. Complicated code is needed to deal with different intermediate values and special code for combining decompression and compression process together. Less calculation will be needed, if we can just insert bits between values.

So another option is to benefit from Intel BMI2 special instructions. As Figure 27 shows, the instruction "pdep" can insert '0's into the values according to the corresponding mask.

With this operation we can alter bitcases from one to the other without going through intermediate values. As Algorithm 10 shows, in line 3, we load 64 bits at a time. Then we combine the value that is left from previous iteration. After that we calculate and create a corresponding mask for

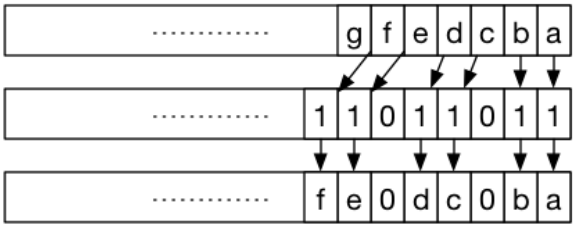


Figure 27: PDEP instruction

this vector, and we use a PDEP instruction to insert the '0's and write data back to memory. An intermeidate buffer is not necessary in this process. However, the instruction operates only on 64-bit registers, so it needs more memory access operations than SIMD rollover.

Algorithm 10 Rollover with BMI instructions

1: set k to 0
2: **for** i from 0 to $max\_index/64$ **do**
3: parallel_load a_1 from input
4: tmp_{a1} <– a_1 << (unused bits b)
5: r <– tmp_{a1} or $left$
6: generate $mask(m_0, ..., m_{63})$
7: PDEP to insert bits into r with $mask(m_0, ..., m_{63})$
8: left <– $a1$ >> (64 - unused bits b)
9: **end for**

Special Algorithm For some bit cases there may be special algorithms. As an example, we convert bitcase 1 to bitcase 2. For convenience, we assume that we are working on one cache line at a time.

As Figure 28, in the first step, the first 32 bytes of a cache line are loaded into the highest and into the lowest half cache line. Then a shuffle step spreads the bytes in order with zeroes inbetween each pair. At this point, every bit is either in the correct byte or in the adjacent one. Now we recursively correct things: with a right shift of 4, a mask, and an xor, we can get packs of 4 bits in the lowest 4 bits of the correct byte. With a right shift of 2, a mask, and an xor, we can split up each pack of 4 bits into two packs of 2 bits with two bits space inbetween (every other value is now in the correct place, followed by its neighbor instead of a zero). Finally, a right shift of 1, a mask, and an xor give us the result (the alternation of 1 input bit and a zero bit).

It should be sufficient to slightly adapt the first two steps to make it work for AVX-2 (working on half a cache line).

A similar principle can be applied for other cases. Rollover bitcase 2 –> 4 and 4 –> 8 work trivially the same by leaving out the last recursive step(s). Other bit cases may be more complicated as we might need shifts across word boundaries, which needs further studies.

5 Evaluation

Last sections, we have proposed SIMD algorithms for compression. And we also propose different several algorithms for rollover. We implemented these algorithms and compare them with each other in this section.

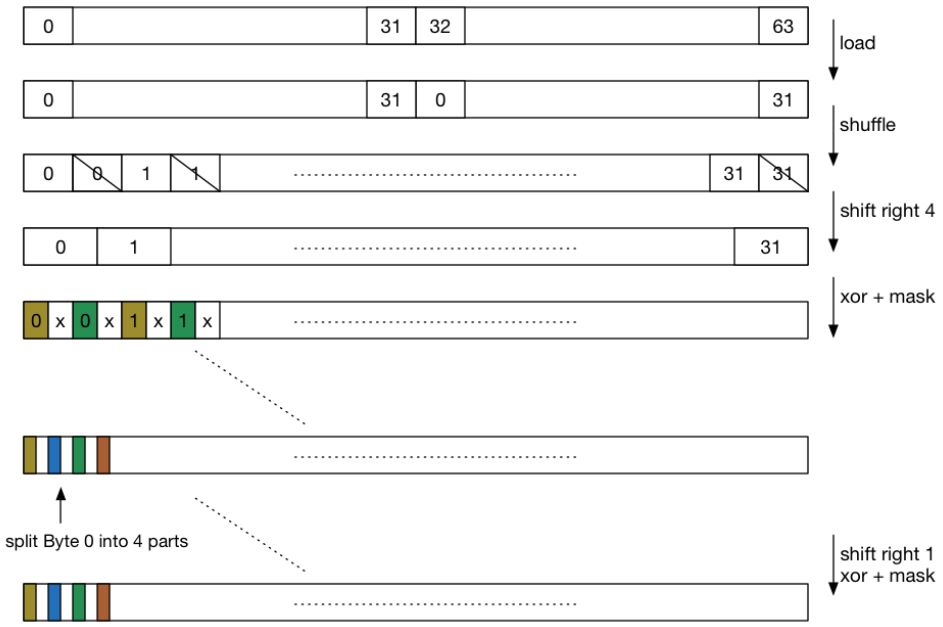


Figure 28: Special Rollover for bitcase 1 –> 2

5.1 Hardware

As we previously described in background Section1.3, our implementation will work both on SSE4 with 128-bit long registers and AVX2 with 256-bit long registers. AVX2 is first introduced in Haswell architecture.
Unless otherwise mentioned, all experiments run on Intel Xeon CPU E5-2697 v3 (based on the architecture codenamed Haswell) with a nominal clock speed of 2.6GHz. CPU information details are listed in Table 3.

CPU Model	Intel Xeon CPU E5-2697 v3 @ 2.6GHz
CPU Frequency	2.6GHz(Turbo Mode up to 3.6GHz)
Architecture	Haswell
Cores per socket	14
L1d/i cache	32K
L2 cache	256K
L3 cache	35MB
Memory	65GB

Table 3: CPU Information Details

Our machine was equipped with SLES11.3 using Linux kernel version 3.0.76.
We use GCC version 4.9.2 to compile the compression algorithm. For rollover, we use the SAP HANA environment with Intel Compiler ICC version 15.0.1 (GCC version 4.3.0 compatiblility). We integrate our code into the SSE library of HANA for testing. For implementing our algorithm, we use intrinsic functions provide by Intel.
For measuring the performance, we use Intel Performance Counter Monitor, which is more accurate than Linux system clock.
Current systems uses NUMA architecture. When running a program, threads and memory will be distributed to different cores, and the access for a thread of one socket to memory of a different socket is slower than access within the same socket. This leads to a high variance in execution time.

For eliminate the effect of transfer data from one core to another, we use “numactl” to pin the program to a specific core.

5.2 Estimating CPU cycles

Before we execute our programs on hardware, we can first analyse them. With data Intel provided by Intel, the number of CPU cycles can be calculated for each bitcase.
We can count cycles both for the one using SIMD instuctions and naive version. CPU cycles for load and store is difficult to estimate, considering the level of cache we get the data from and the cache misses. Therefore we only estimate CPU cycles for computing. The compute cycles for different instructions differ from Intel architectures. Let us take the newest architecture Haswell, as example. Table 4 lists the theoretical compute cycles of the each instructions. We take bitcase 17 using 128 bits vector (SSE instrution) as an example to estimate the compute cycles.

operator	theoretical cycles
shift	1
shuffle	1
or/and	1
permute	3
extract	2

Table 4: Latency of Instructions on Haswell [Int14]

operator	Generic	SSE	AVX
shift	188	126	80
shuffle	0	64	32
or+and	124	158	111
permute(avx only)	0	0	16
extract	0	68	0
load	128	32	16

Table 5: Numbers of operations to uncompress 128 values of bitcase 17

	Generic	SSE	AVX
compute CPU cycles per value	2.4375	3.78125	2.1171875

Table 6: CPU cycles per value for compressing 128 values of bitcase 17 without load & store

We also calculate the CPU cycles we have used in the process. Assuming that “load” operation takes 2 CPU cycles to get data from level 1 data cache.
To compare the different algorithms in Figure 29, we compare the estimated CPU cycles between the SSE version and the AVX version of the algorithm. The SSE version needs twice as much “load”s as the AVX version. It takes more “or/and”, “shift” as we have expected. However, the SSE version uses an “extract” instruction to get the first 32 or 64 bits from a SIMD register. Each “extract” instruction takes 3 CPU cycles to execute. For the AVX version, besides the in-lane operations, it still takes time to exchange data betweeen lanes.

operator	Generic	SSE	AVX
shift	64	0	0
shuffle	0	32	8
or+and	64	158	111
permute(avx only)	0	0	8
extract	0	32	0
load	128	32	16

Table 7: Number of operations to compress 128 values of bitcase 16

	Generic	SSE	AVX
compute CPU cycles per value	1	1	0.25

Table 8: CPU cycles per value for compressing 128 values of bitcase 16 without load & store

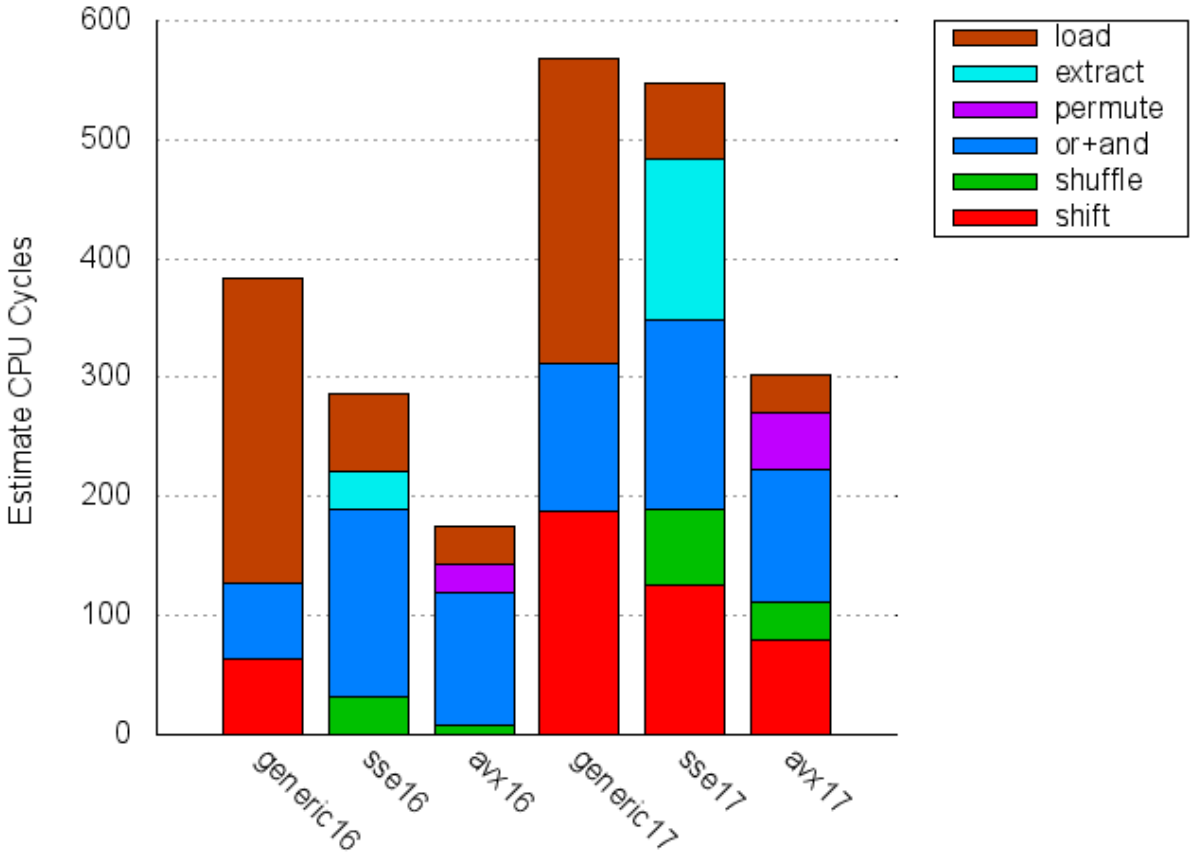


Figure 29: Analysis and comparison of SSE implementation with AVX for bitcase 16 and bitcase 17

For comparing the different bitcases of SSE version, we calculate the instructions used in bitcase 16 and bitcase 17. Bitcase 16, which is byte aligned, benefits from "shuffle". There is no "shift" instruction in this process, and it does not require saving the remaining bits from the last iteration.
For the AVX version, all in-lane operations are the same. For bitcase 16, besides the shuffle, we still need a "permute" to exchange values across lanes.
From Table 8, we can easily come to the conclusion that the SSE version of compression uses more operations per value than the scalar version. AVX each has twice the length of SSE which solves the problem. Both SSE and AVX save a considerable amount of time via less load and store operations, so it is worth to be implemented.

5.3 Test Result

One SSE register is 4 times the size of an integer, and AVX2 is 8 times the size. Theoretically, an SSE version of a program should be 4 times faster, and AVX2 8 times. As it is shown in the work by Willhalm et al. [WBP+09] and [WOMF13], the decompression process with SSE has 1.5-3 speedup, as Figure 30 shows, with AVX2 being 30% faster than SSE version.

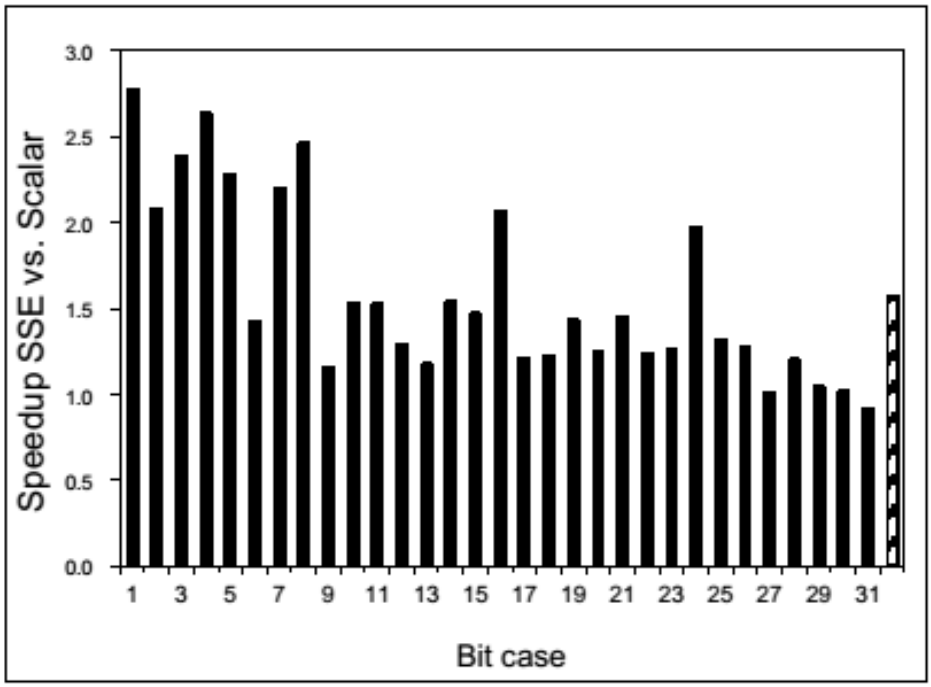


Figure 30: Unpacking Speedup with SSE [WOMF13]

5.3.1 Compression Test Result

The compression algorithms are more complex than the decompression algorithms. From Figure 31, it is easy to see that we get less benefit from SIMD instructions than in the decompression process. Figure 31 also depicts that, for smaller bitcases, there is not much difference between AVX and SSE, because the register utilization is low. As we have calculated in Table 6, SSE needs more operations for calculation than the AVX2 algorithm does. So for larger bitcases, AVX outperforms SSE.
For bitcase one, we use special instructions. With that we have up to 2.4 speedup. If Intel provided special instructions for all bit cases, it is certain that there is a huge potential to make the compression process even faster.
Other special case such as 4, 8, 16, 24, which we use have optimized with shuffle instructions, do not show much difference in speedup. We also notice that speedup prove the analysis. By processing less

values at a time, SSE version lost a lot of performance.

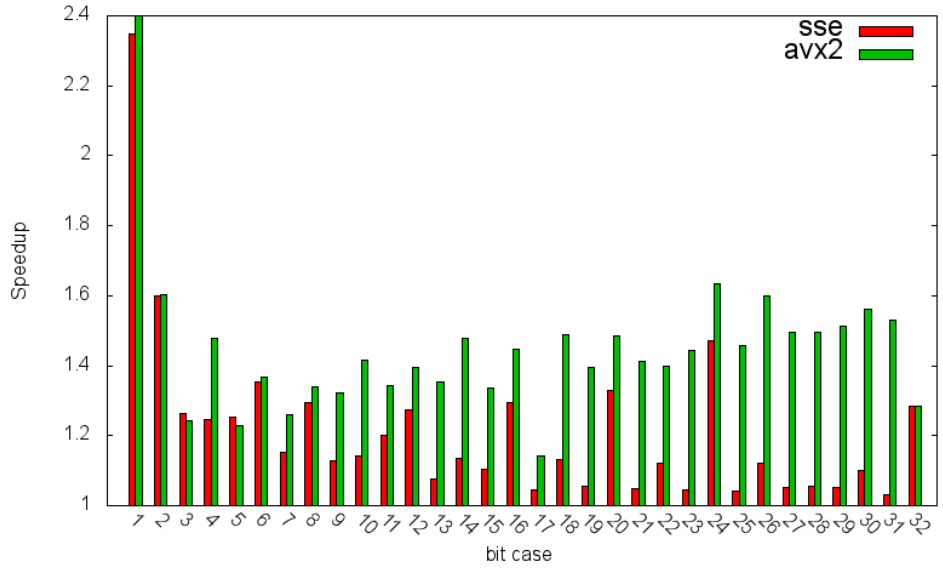


Figure 31: Speedup, comparing SSE and AVX version with scalar compression

Figure 32 depicts the execution time. Because they use the same algorithm, the execution time of SSE4 and AVX2 are showing the same pattern of performance peaks.
Firstly, the execution time of the naive algorithm grows slower than both SIMD versions, because in the naive version, we take on average 1 “or” and “shift”, while we optimized SIMD version for processing the smaller bitcases in parallel.
Secondly, it also illustrates that in smaller bitcases, SSE and AVX2 don’t show much difference. From bitcase 8 to bitcase 31, the AVX2 algorithm outperforms SSE.
Another thing worth noticing is that we have particular good performance at bitcases 4, 8, 16, 24, because we can benefit from special shuffle instructions for these bitcases. SSE4 and AVX2 flags also have been setup for compiling scalar version. The Compiler also optimizes these bitcases in some degree. By improving the compiler, the scalar algorithm may also get better performance.
For bitcase 32, as we have illustrated, it only executes a memcpy. So the speedup we have in the figure is only the benefit from non-temporal write.
In Figure 33, we compare the scalar version and the scalar verion with Non-Temporal Stores. For larger bitcases, compression is approximately 10% faster with Non-Temporal Store technique. Bitcase 1, 4, 16, 24, 32 are exceptions that we have noticed, non-temporal write has bad performance in bitcase 4, but particular good performance in bitcase 1, 4, 16, 24, 32. For these bitcases we need further studies.

5.3.2 Rollover Test Result

Test result of the compression process proved that we can certainly benefit from vectorization. And we have also implemented the different algorithms of rollover. The algorithms we have tested contain: naive version, blockwise version, SIMD version and PDEP. We have tested bitcase 17 with 10^9 Value-IDs. We repeat the test for each algorithm 10 times, and we use the execution time average of 10 iterations.
For special algorithm, research is still needed for processing other bitcases. Therefore, we have not implemented the special algorithm. The comparison of other algorithms show in Figure 34, the naive algorithm takes the longest time, and has a large memory consumption. The Blockwise algorithm is the easiest to implement, and it can achieve slightly better result. The SIMD version is the most complex to implement, because it combines SIMD decompression and SIMD compression together with a new interface, and still translates the values into intermediate values. PDEP version benefits from direct operation on the value from original to the target. Therefore, it saves both the time to translate values

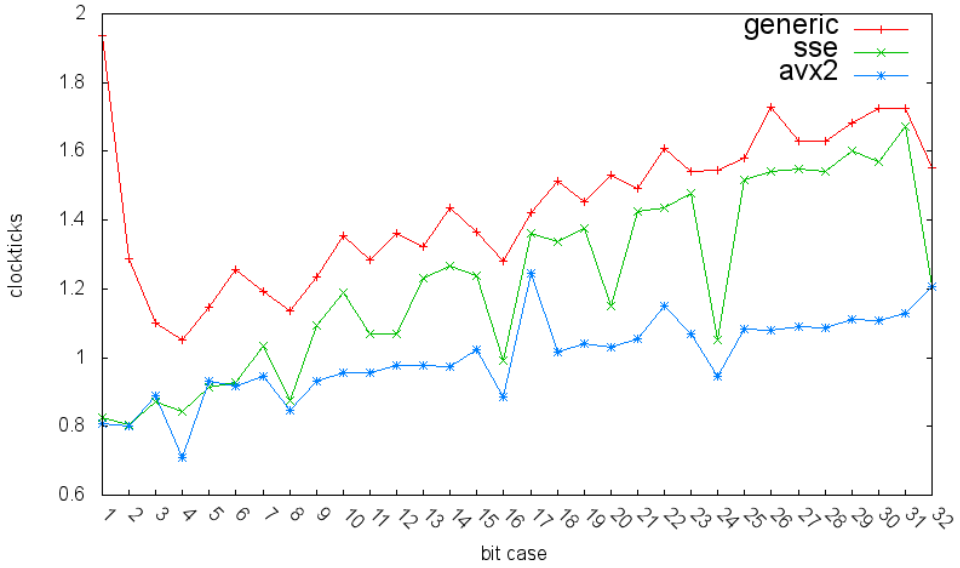


Figure 32: Compression: execution time of all bitcases in clock ticks (per values)

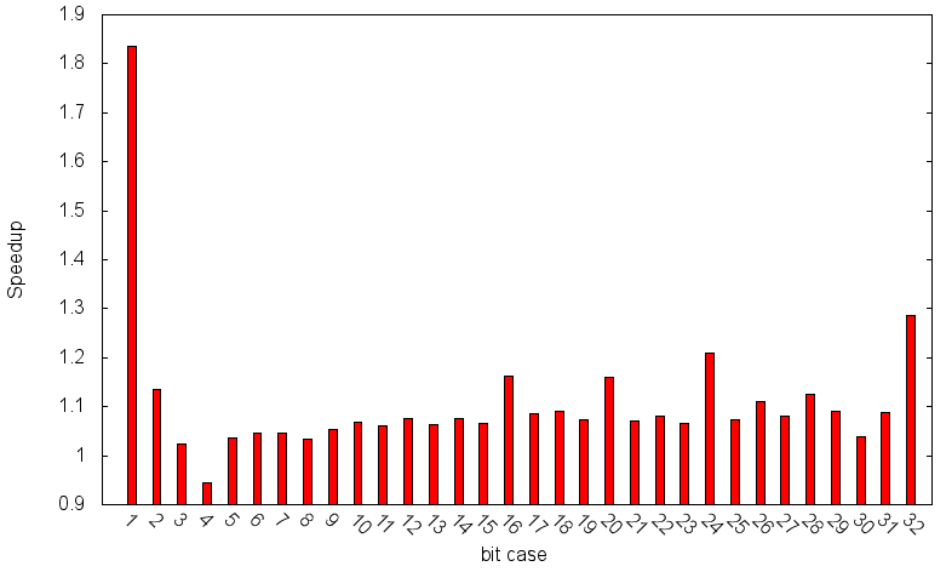


Figure 33: Comparing scalar compression and scalar compression with non-temporal store

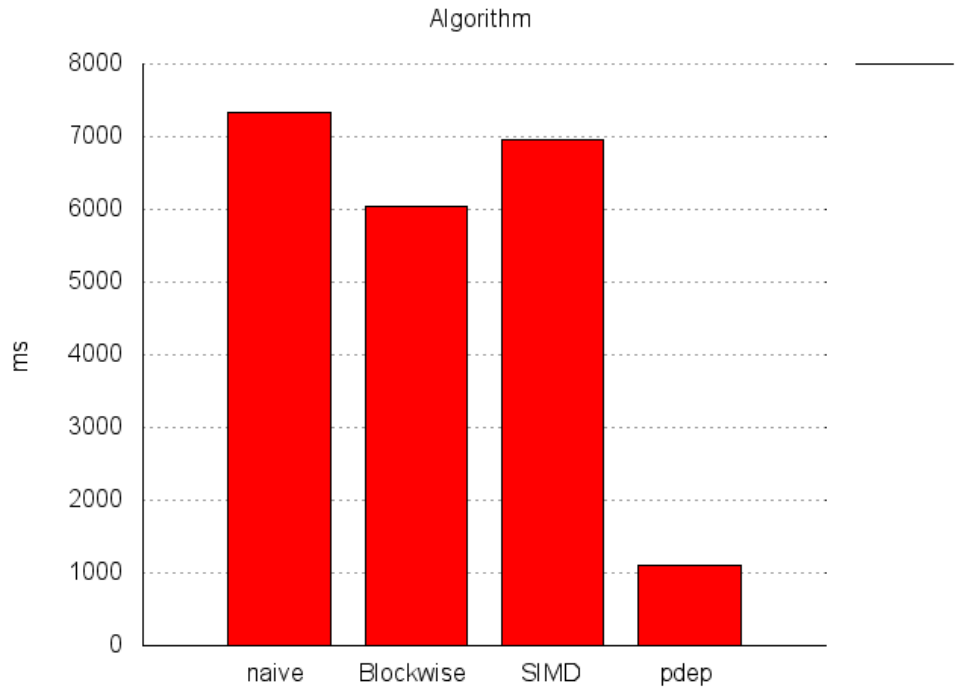


Figure 34: Rollover execution time of different algorithm, 10^9 values, 10 iterations

to intermediate values, and the intermediate buffer (memory, cache, register) used by the previous three algorithms. According to Table 9, PDEP version has up to 6.64 speedup for bitcase 17. For larger bitcases, PDEP version process less values in one iteration, therefore may get less speedup. Further more, SIMD version can be easily adapt to process remapping of the dictionary, which PDEP version can not.

algorithm	execution time(ms)
naive	7342
Blockwise	6052
SIMD	6968
PDEP	1105

Table 9: Rollover execution time of different algorithm for 10^9 values, 10 iterations

Assuming Intel will release a 128 bit or a 256 bit version of PDEP, the rollover process can be further accelerated.

5.4 Summary

In this section, firstly, we have analyzed the bitcase 16 and bitcase 17 of the SIMD version of compression, which belongs to different classes of algorithms. During the analysis, we have compared between different algorithms. We draw the conclusion that bitcase 16 can benefit from certain instructions because bitcase 16 is byte-aligned. We have also compared the SSE and the AVX version. We have shown that AVX is faster than SSE, because it has wider registers.
After an analysis, we have provide test results of all bitcases. We illustrated the result both with execution time and speedup.
For the last part, we also tested different algorithms of Rollover. The test result shows that the SSE and

the AVX versions of rollover are not worth to be implemented. A better way to implement Rollover special 64 bits instruction PDEP to generate the result directly without going through the intermediate values.

6 Conclusion and Outlook

According to Moore's law, memory size in standard servers have increased every year. For solving the bottleneck of data transfer from disk, In-Memory database systems, which keep all relevant data in memory, has been introduced to overcome the problem. Data separates into a compressed main storage for read operations and an uncompressed differential buffer. And differential buffer will be merged into main storage in a certain time. Because merge operation is frequently executed, efficient decompression and compression algorithm need to be implemented.
Vectorizing the decompression algorithms has been introduced in previous work [WOMF13] and [WBP+09], one with SSE instructoins and another with AVX.
In this thesis, first and foremost, we explore the compression algorithm with Intel SIMD instruction sets, SSE and AVX2. We have introduced 5 algorithms according to bitcases: Bitcase 1, Bitcase less than 8, Bitcase multiples of 8, Bitcase 32, and Bitcase 9-31 except 16 and 24. The first 4 algorithms are special bitcases, where we can benefit from certain instructions. For the last group, we have introduced with the 2 different algorithms: the first has problems with saving the remaining bits from the last iteration, the second benefits from an AVX2 instructions that shifts packed values with different offsets. With the second algorithm, we saved operations on dealing with the bits we cannot store to the memory in the current iteration.
We have also introduced a process called Rollover that combines the decompression and compression together. We list two scenarios where we need rollover, and we introduced 5 different algorithms: a naive algorithm, a blockwise algorithm, a SIMD algorithm, an algorithm with PDEP instruction, and a special algorithm.
After that we analyse the code segment of compression algorithms, which we have implemented. From that we compare the different bitcases, and we compare the same bitcase with different instrution sets.
Finally, we test implemented our compression algorithms and rollover algorithms. From the result we can tell the advantages and disadvantages of each algorithm.
From all above, we get to the conclusion that the merge process can benefit from using SIMD instructions, which has wider registers.
There are still several points need to be studied in the further works:

- In compression Section 3, we have introduced an alternative compression algorithm for the currently implemented one, Algorithm 5. This algorithm is introduced to solve the shift boundaries in Intel SIMD instructions, which can simplify the operations for saving the bits from the last iteration. Yet it needs further research to prove its efficiency.
- By using Non-Temporal writes, we have noticed that there are several bitcases with odd results. We have not found any explanation for these bitcases.
- In Section 4, we have introduced a special algorithm, which currently only works to extend bits from 2^b to 2^{b+1}. Also, we have not evaluated this algorithm.
- In Section 4, we have also discussed that after merging the values, Value-IDs vector need to be remapped. Remapping the Value-ID vector also cost tremendous execution time, which can possibly be vectorized with Intel SIMD instructions.

Acknowledgements

I offer my sincerest gratitude to Prof. Wolfgang Lehner from TU Dresden for the discussion and suggestion for the thesis. I would like to thank my project supervisor Reza Sherkat from SAP HANA for

supporting with his expertise in this region, and my academic supervisor Ingo Müller also from SAP HANA for giving me suggestion about the thesis and the algorithms. Finally I would also like to thank Intel expert Dr. Thomas Willhalm for helping understanding the Intel instructions and testing the implementations.

References

[FGB+14] FAUST, Martin ; GRUND, Martin ; BERNING, Tim ; SCHWALB, David ; PLATTNER, Hasso. Vertical Bit-Packing: Optimizing Operations on Bit-Packed Vectors Leveraging SIMD Instructions. Springer-Verlag Berlin Heidelberg. 2014

[Flo14] FLORENDO, Colin: IQ Real Time PlexIM Merge Design. Dublin, California: Sybase Inc., 2014

[Int14] INTEL CORPORATION: Intel Architecture Instruction Set Extensions Programming Reference. 2200 Mission College Blvd. Santa Clara, CA 95054-1549 USA: Intel Corporation, 2014

[KB14] KLEIS, Wolfram ; BENDELAC, Chaim: SAP Architecture book. Dietmar-Hopp-Allee 16, Walldorf: SAP SE, 2014

[LB13] LEMIRE, Daniel ; BOYTSOV, Leonid. Decoding billions of integers per second through vectorization. Software: Practice & Experience. 2013

[SAB+05] STONEBRAKER, Mike ; ABADI, Daniel J. ; BATKIN, Adam ; CHEN, Xuedong ; CHERNIACK, Mitch ; FERREIRA, Miguel ; LAU, Edmond ; LIN, Amerson ; MADDEN, Sam ; O'NEIL, Elizabeth ; O'NEIL, Pat ; RASIN, Alex ; TRAN, Nga ; ZDONIK, Stan. C-store: A column-oriented DBMS. 2005

[WBP+09] WILLHALM, Thomas ; BOSHMAF, Yazan ; PLATTNER, Hasso ; ZEIER, Alexander ; SCHAFFNER, Jan. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. VLDB. August 2009

[WOMF13] WILLHALM, Thomas ; OUKID, Ismail ; MÜLLER, Ingo ; FAERBER, Franz. Vectorizing Database Column Scans with Complex Predicates. ADMS 2013, Riva del Garda, Italy. August 2013