SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units

Thomas Willhalm
Nicolae Popovici

Intel GmbH
Dornacher Strasse 1
85622 Munich, Germany
thomas.willhalm@intel.com
nicolae.o.popovici@intel.com

Yazan Boshmaf

SAP AG
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
yazan.boshmaf@sap.com

Hasso Plattner
Alexander Zeier
Jan Schaffner

Hasso-Plattner-Institute
University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
office-plattner@hpi.uni-potsdam.de
alexander.zeier@hpi.uni-potsdam.de
jan.schaffner@hpi.uni-potsdam.de

ABSTRACT

The availability of huge system memory, even on standard servers, generated a lot of interest in main memory database engines. In data warehouse systems, highly compressed column-oriented data structures are quite prominent. In order to scale with the data volume and the system load, many of these systems are highly distributed with a shared-nothing approach. The fundamental principle of all systems is a full table scan over one or multiple compressed columns. Recent research proposed different techniques to speedup table scans like intelligent compression or using an additional hardware such as graphic cards or FPGAs. In this paper, we show that utilizing the embedded Vector Processing Units (VPUs) found in standard superscalar processors can speed up the performance of main-memory full table scan by factors. This is achieved without changing the hardware architecture and thereby without additional power consumption. Moreover, as on-chip VPUs directly access the system's RAM, no additional costly copy operations are needed for using the new SIMD-scan approach in standard main memory database engines. Therefore, we propose this scan approach to be used as the standard scan operator for compressed column-oriented main memory storage. We then discuss how well our solution scales with the number of processor cores; consequently, to what degree it can be applied in multi-threaded environments. To verify the feasibility of our approach, we implemented the proposed techniques on a modern Intel multi-core processor using Intel® Streaming SIMD Extensions[1] (Intel® SSE). In addition, we integrated the new SIMD-scan approach into SAP® Netweaver® Business Warehouse Accelerator[2]. We conclude with describing the performance benefits of using our approach for processing and scanning compressed data using VPUs in column-oriented main memory database systems.

VLDB '09, August 24-28, 2009, Lyon, France.

1. INTRODUCTION

Computer technology is continually developing, with abiding rapid improvement in processor architecture, disk storage, and main memory capacity. At the same time, the massive increase in data volumes has created a demand for high performance data management capabilities. This is reflected by the data-intensive query processing tasks like OLAP, data mining, and scientific data analysis. These tasks rely on powerful hardware resources and require optimized software solutions for processing the huge amount of data with high performance.

As the system memory gets larger and cheaper, database systems started therefore to evolve from disk-based to memory-based operation (and storage). As a result, main memory is becoming a critical resource. As in disk-based database engines, data compression techniques are considered as one way to handle this new main memory bottleneck.

Previous research showed that the performance of relational disk-based database system can be increased by extending the storage manager, the query execution engine, and the query optimizer to handle compressed data [1].

In main memory column-store database systems like SAP® Netweaver® Business Warehouse Accelerator (BWA), relational tables are completely loaded into memory and are stored column-wise. In order to save RAM and to improve access rates, the in-memory data structures are highly compressed. This is achieved by using different variants of Light Weight Compression (LWC) techniques like run-length encoding or multiple version of fixed-length encoding.

In SAP® Netweaver® BWA, the default compression mechanism is dictionary compression with bitwise encoding of columns. Here, all distinct values of a column are extracted, sorted, and inserted into a dictionary. The column itself keeps only references to the dictionary, where each reference value is stored using as many bits as needed for fixed-length encoding.

While most access functions work directly on compressed data by implicit decompression, operations like projection have to materialize the data, and therefore, explicitly decompress it. Here, high performance is achieved by making optimal use of the CPU local cache. This is accomplished by processing small data chunks in one execution step. Hence, data decompression is becoming a significant part of the query execution.

In database engines similar to SAP® Netweaver® BWA, the central low-level access function is the main memory full table scan over highly compressed data (table columns). This operation requires lots of calculations and is CPU-intensive. The need for decompression increases the CPU-bound behavior, whereas a scan over uncompressed data is more memory-bus-bound.

With the arrival of multi-core processors, operating on compressed data is continuously becoming cheaper as CPU processing rates are increasing faster than data-access bandwidth rates [2]. As a result, more sophisticated compression algorithms are being used while full table scan operations are shifting from being IO-bound to CPU-bound. In addition, the data in the system's memory is being compressed as much as possible. This created a new exciting research field focusing on optimizing table scan operations for multi-core processors using different parallelization techniques. However, even though multi-core processors offer rich Simultaneous Multi-Processing (SMP) experience, their fast Vector Processing Units (VPUs) have not been fully exploited to vectorize and streamline table scan operations.

In this paper, we introduce a novel SIMD approach for in-memory fast table scan operations working on compressed table columns. We utilize the latest SIMD capabilities of each core in superscalar multi-core processors to efficiently decompress in-memory table columns and search for a scan value with a considerably lower latency. With this approach, we extend the parallelization of table scan operations that has been limited to multithreading on the task and data level.

SIMD, firstly classified by Flynn [3][4], represents a vector processing model providing instruction level parallelism. SIMD forms an important extension to modern processors architectures and provides the ability for a single instruction stream to process multiple data streams simultaneously. Figure 1 shows the SIMD execution model.

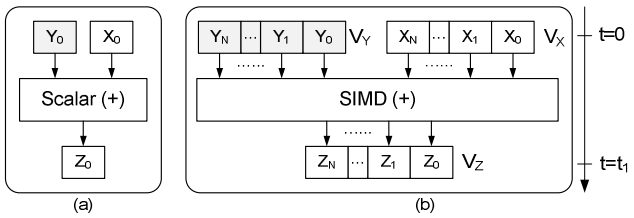

Figure 1. SIMD execution model: In (a) scalar mode: one operation produces one result. In (b) SIMD mode: one operation produces multiple results

1.1 Contributions

Full table scans are the basic operations in main memory column-store database systems. They are heavily used in standard query execution as the system tries to avoid creating indexes in order to reduce memory consumption.

In this paper, we propose a new SIMD approach to execute the following in-memory table scan operations in a very short latency (as shown in Figure 2):

- **Vectorized Value Decompression:** During a scan operation, column values might have to be explicitly decompressed in order to eventually continue the query execution in operations like projections. We introduce a fast SIMD decompression approach for the LWC Number Compression (LWC-NC) technique that is widely used in today's DBMSs. Nevertheless, the approach is also feasible for other compression techniques.
- **Vectorized Predicate Handling:** During typical table scan operation, simple predicates like equal-value or value-range search queries must be executed. We present a new concept and implementation of SIMD predicate search. Here, we adopt the concepts in our decompression approach to realize a vectorized value search process without the need to decompress the actual column values.

As full table scan is the basic function for many highly advanced data management operations (like aggregation) or complex predicates, our approach is feasible for nearly all kinds of database operations. For compressed data structures, which are common in in-memory databases, adding numbers together is negligible compared to decompression. As a result, we measure and show that effect in our experiments (i.e. decompression and full table scan).

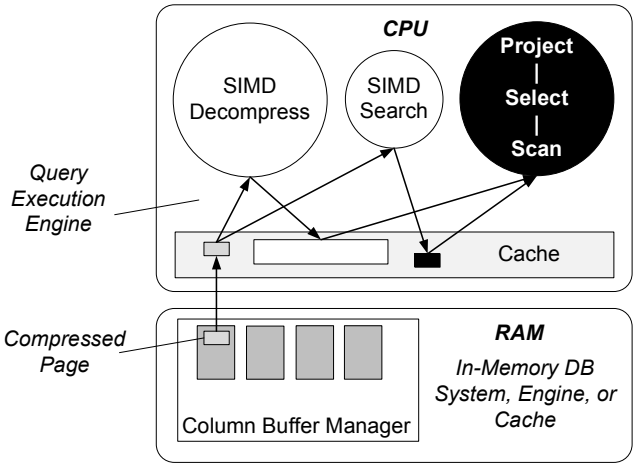

Figure 2. Vectorized table scan operations

Besides, we show that vectorizing the scan operations to utilize the embedded VPUs achieves a significant performance gains and reduces the overall system-resources consumption. As well, additional power requirements and expensive data movements between the system's main memory and external accelerators (like graphics cards, FPGAs, or other coprocessors) are avoided.

We also show that our approach scales well with the number of processor cores. This can be accomplished because our approach is completely thread-able and can be parallelized at the task and data level, and thus, it exploits the SMP environment offered by multi-core processors.

Paper Outline: The remainder of this paper is organized as follows: After discussing relevant background information in Section 2, we describe the system model in Section 3 and the main concepts of our approach in Section 4. This is followed by a detailed description of the implementation in Section 5. After that, we present our real-world evaluation results in Section 6. Finally we conclude the paper in Section 7.

2. BACKGROUND

In this section, we reference related work and its connection to ours. After that, we present background concepts related to this paper. We give an overview of the light-weight main memory database compression and then present a typical SIMD execution model of modern VPUs.

2.1 Related Work

Database compression research focuses mainly on two aspects: First, compression algorithms and their integration into database systems. Second, compression performance aspects such as query response time speedup and disk-storage savings achieved. The authors in [1] concentrated on studying the performance of database compression techniques, specifically Light-Weight Compression (LWC), and recommended their integration into relational database systems for increased performance.

Performance improvements and speedups achieved by utilizing data compression in query processing are outlined in [5]. There, the authors illustrated a novel idea of leaving the data compressed as long as possible and only decompressing it when utterly needed. Also, they showed how query processing algorithms can be extended to process compressed and decompressed data. This resulted in query processing speed up by a factor considerably larger than the compression ratio.

New versatile LWC compression variants that are specifically designed for optimized Instruction per Cycle (IPC) of modern CPUs are proposed in [6]. These new compression techniques target both LWC Number Compression (LWC-NC) and LWC Dictionary-based Compression (LWC-DC) and provide an improved compression ratio, RAM/CPU cache compression, and are superscalar friendly (i.e. no if-then-else constructs, loop pipelineable, and allow out-of-order execution).

Thread-level parallelism for database compression is discussed in [7]. In that paper, the authors discuss compression and bandwidth tradeoffs for database scan operations. They also explore the parallelization of the decompression task at the data level and showed that splitting compressed tuples into compression blocks for each thread is efficient. They also noticed that even though the processing of difference coded tuples within a block is serialized, there is inherently a good parallelism across blocks.

Reducing the current RAM-bound performance and optimizing table scan operations are addressed in [8] and [9]. In [8], the memory access serialization issue created by having many active threads that are pinned to separate processor cores is handled by sharing table scans by multiple queries. This is achieved by having a new data structure that keeps track of multiple aggregations in multiple queries and grouping as many queries as possible in each core's local cache. Alternatively, a new layout and processing technique for efficient one-pass predicate evaluation is proposed in [9]. There, the authors propose a new way to evaluate predicates that is highly parallel compared to the column by column partial evaluation in pure column-store database systems.

Using SIMD technology to vectorize database processing has received a significant attention in the database community, in particular for the applicability of SIMD instructions for database workloads [10]. A study on the suitability of the Cell Broadband Engine for database processing was conducted in [11] by vectorizing all operations and eliminating performance-critical branches while being restricted to a small program code. However, the authors only discuss the applicability of vectorized database management (hashing, in specific).

No recent research, to our knowledge, has touched upon vectorizing LWC (de)compression techniques or table scan operations using SIMD technology on embedded VPUs.

2.2 Light-Weight Database Compression

There are many LWC algorithms and techniques that can be used in database systems but a small number of them have proved high performance with respect to response time and overall query latency. Such algorithms are Numeric Compression (NC), String Compression (SC), and Dictionary-based Compression (DC), to name few.

In this paper, we focus on LWC compression for its high granularity and low-latency. In general, LWC is applicable to a whole file in the database (i.e. a relation of a partition of a relation), a page of a file, a tuple, or any field in the tuple. The highest degree of granularity is field-level compression which means that every field in every tuple in the database can be compressed or decompressed without any dependency on (i.e. reading or updating of) fields in the same or other tuples. Additionally, LWC is very fast in terms of CPU-bound latency when compressed and decompressed. This is very important as most of the database operations are CPU-intensive (joins and aggregations, for instance) with small CPU time left for other operations. This can be explained by the continuing trend towards calculation-intensive systems (like business intelligence solutions) that lead to more CPU-bound scenarios. Practically, modern processors provide sufficient memory bandwidth to make even a simple search CPU-bound, if the data is only lightly compressed.

LWC-NC compression of integers is based on null suppression and encoding of the resulting length of the compressed integer [12]. For example, the integer value "3_d" can be stored by storing only "11_b" and ignoring the other thirty "0_b" bits.

Formally, the storage of a contiguous unsigned integer array D with a maximum integer value of m can be achieved by using only n bits for each integer (Eq. 1). Also, using this concept leads to a compression ratio r of the array D as described in Eq. 2. Typically, $sizeof(int)$ equals 32 bits but this value can change as the physical representation of integer data types may differ depending on the system architecture.

$$n = \lceil Log_2^m \rceil \qquad \dots (\text{Eq. } 1)$$

$$r = \frac{n}{\text{sizeof}(int)} \qquad \dots (\text{Eq. } 2)$$

For example, an array of 1024 unsigned integers would normally require 4 Kbytes (1024*32 bits). Supposing a maximum integer value of "511_d", then all integer values can be stored using only 9 bits with a total size of 1.125 Kbytes. By doing so, 2.875 Kbytes are saved - 72% of the original storage size.

The previously described LWC-NC model represents a generalization of a set of different variants of LWC-NC compression techniques like Prefix Suppression and Frame of Reference compression.

In Prefix Suppression (PS) [5], the data is compressed by neglecting common prefixes in data values. Typically, this is done in the special case of zero prefixes for numeric data types. Thus, PS compression can be used for numeric data if actual values tend to be considerably smaller than the largest value of the type domain.

Frame of Reference (FoR) compression [13] keeps for each disk block the minimum min_C value for the numeric column C, and then stores all column values $c[i]$ as $(c[i] - min_C)$ in an integer of only $\left\lceil Log_2^{(max_C - min_C + 1)} \right\rceil$ bits. FoR compression is efficient for storing clustered data (dates in a database, for instance) as well as for compressing node pointers in B-tree indices.

FoR compression resembles PS if $min_C = 0$, though the difference is that PS is a variable-bitwidth encoding, while FoR encodes all values in a page with the same amount of bits.

In this paper, we propose a fast SIMD approach for modern superscalar CPUs to decompress LWC-DC data using the generalized model described earlier in Equation 1 and Equation 2. Our discussion targets compressing unsigned integers using a fixed-bitwidth encoding but can be easily extended to include different data types (signed or unsigned) and for both fixed and variable-bitwidth encoding. For example, variable-bitwidth encoding can be handled by intelligently packing and compressing each bitwidth-case in a separate memory block. Now, decompressing these blocks is achieved by running our approach against them, each as a fixed-bitwidth encoding. To clarify, if a set of database indexes is compressed using a mix of 4-bit, 8-bit, and 16-bit number encodings, then three compressed sets are generated each having a fixed-bitwidth encoded values. These compressed sets can be seamlessly decompressed (in parallel) using the concepts we introduce in our approach.

2.3 SIMD Execution Model

We assume a powerful and versatile implementation of SIMD that provides a rich set of integer and floating point instructions. These SIMD features increase the overall performance, execute one n-bit multi-operand operation in a single cycle (n = {32, 64, 128 …}, 4 operands for 32-bit SIMD instruction in a 128-bit SIMD execution model, for instance), and hence improving the energy efficiency by doing more computations in less time.

Typically, there are four vectorization methods supported by modern compilers: inline assembly language, intrinsics, language class libraries, and finally automatic vectorization. Each method represents a tradeoff between controllability and usability, where automatic vectorization by the compiler being the most usable but with the least degree of control.

Conceptually, the SIMD instruction-set should provide sufficient options to efficiently perform the following operations for a set of input operands:

- **Arithmetic Operations:** Addition, Multiplication, etc.
- **Logic Operations:** ANDing, ORing, Shifting, etc.
- **Compare Operations:** String compare, Block compare, etc.
- **Data Movement:** Load/Store, String copy, Block copy, etc.
- **Miscellaneous Operations:** Data-type conversion, Shuffling, Concatenation, Cache-ability, etc.

3. SYSTEM MODEL

For the sake of discussion, we assume a SIMD implementation that works on 128-bit registers, provides a signed/unsigned 8/16/32/64/128-bit operand operations as discussed in Section 2, and is independent from the normal scalar execution core (i.e. they can execute in parallel). The discussion in this paper can be easily applied to SIMD implementation with different registers sizes. Figure 3a illustrates the SIMD register model.

We also assume that all integer data (either compressed or not) are stored in a byte-accessible memory using a contiguous address space. Typically, in 32-bit system architectures, each integer requires 32 bits to be stored. As a result, n-number of integers are stored in n 32-bit Double-Words (DW) starting at address a' and ending at address $a' + (4 * n)$ and occupying $32 * n$ bits. In 64-bit architectures, each 64-bit Quad-Word (QW) can hold two 32-bit integers. In this paper, we assume 64-bit little-endian architecture with indexing starting at a and increasing by one for each 64-bit. For example, the index $(a + 4)$ points at the first bit of the fifth QW where 32.5 Bytes (4*64-bits) of the memory are skipped. Also, this allows us to access two 64-bit integers as one 128-bit integer. Figure 3b shows the memory layout model.

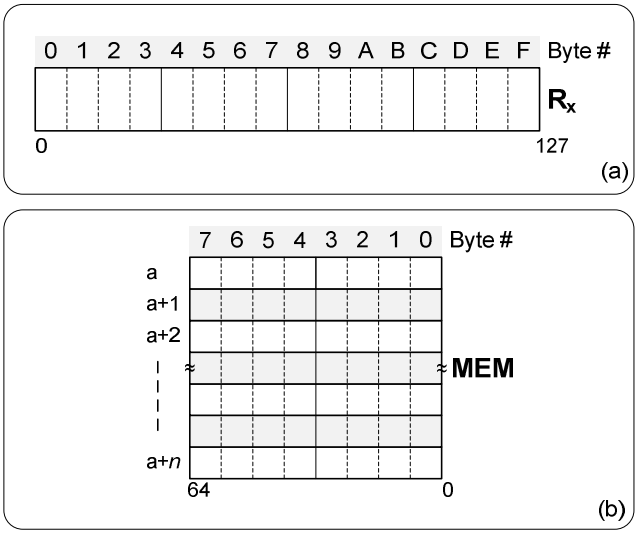

Figure 3. System model: (a) the SIMD register model, (b) the memory layout model

4. CONCEPTS

In this section, we discuss the concepts of our two main contributions. We use the system model described in Section 3 to illustrate these concepts. Also, we use the Light-Weight Compression – Number Compression (LWC-NC) as a default compression technique for an array of unsigned integers as outlined in Section 2.2.

4.1 Vectorized Value Decompression

Here, we propose a novel SIMD approach that is optimized for fast in-memory decompression of LWC-NC compressed data. We also discuss advanced issues related to unaligned data access. In general, vectorized LWC-NC integer decompression into 32-bit equivalent can be divided into three main sequential steps: 16-

Byte Alignment, 4-Byte Alignment, and Bit Alignment to handle loading, copying (shuffling), and extracting (shifting) and storing the compressed values in sequence. All compression-bit cases follow the same steps but may have to be executed differently. We use 9-bit compression in our discussion as most of the n-bit compression cases follow the exact same discussion. Exceptions are noted out and are illustrated and visualized in figures. The general *SIMD-Decompression Algorithm* works as follows:

set k **to** 0
for i **from** 0 **to** $max\_index/128$ {
  for j **from** 0 **to** 15 {
    parallel_load b_a **from** $input[k*16+j*n]$
    shuffle b_a **to** c_a **using** $shuffle\_mask(m_{a0},\dots,m_{a15})$
    parallel_shift c_a **by** $(s_{a0},\dots,s_{a3})$
    parallel_store c_a **in** $output[i*16+j*8]$
    parallel_load b_b **from** $input[k*16+j*n+n/2]$
    shuffle b_b **to** c_b **using** $shuffle\_mask(m_{b0},\dots,m_{b15})$
    parallel_shift c_b **by** $(s_{b0},\dots,s_{b3})$
    parallel_store c_b **in** $output[i*16+j*8+4]$
  }
  increase k **by** n
}

Where n denotes the maximum number of bits as computed by Equation 1. The variable max_index referes to the last index of the $input$ array where $max\_index = sizeof(input)$ if the array is 0-indexed. Also, this index is divided by 128 as we are processing 128 bits a time. The variables b_a, c_a, b_b, and c_b are vector variables holding four LWC-NC compressed integers. The shuffle masks m_a and m_b as well as the shift amounts s_a and s_b depend only on n but not on j and are therefore provided as constants to the algorithm. The following subsections discuss the details of the three steps that are required to realize the algorithm.

4.1.1 16-Byte Alignment

In this step, 128 bits of compressed data are read from the memory at QW-aligned address a and loaded in one 16-byte SIMD register R_L. The number of copied compressed integers depends on how many bits are used to represent each (i.e. the compression bits). For example, if 9-bit compression is used, then R_L would hold 14 compressed integer values. Note that the 15th value is partially copied, and hence, it can't be decompressed using R_L. This step is executed in one 128-bit SIMD load instruction. Figure 4 visualizes the 16-byte alignment (load) step.

In this figure, it should be noted that the next group of values to be loaded in the new R_L are located at address $a'+16$ (i.e. at index $a+2$) where the first integer value is incomplete as it spans between two 64-bit memory QWs at indexes a and $a+1$. One way to correctly load the data, starting at the 15th value, is to index the memory at address $a'+15$ with the first 6 bits being invalid (i.e. the last 6 bits of $v13$). However, this will result in an unaligned SIMD memory access which is very expensive to execute in some architectures. Advanced SIMD implementations offer the two SIMD memory access variants at the same cost (i.e. the same CPI). Another solution is to concatenate the two registers (i.e. the old R_L and the new R_L) and then to shift the result, starting from the old R_L data, by the amount of bytes that keeps the first byte holding the 15th element at the beginning of the result register (15 bytes in Figure 4). Practically, this will result in one aligned 256-bit SIMD load instruction and one 256-bit SIMD shift instruction.

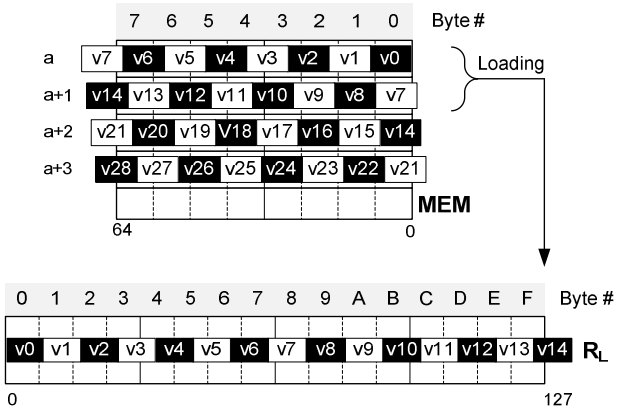

Figure 4. Loading 128-bit compressed block to a SIMD register (9-bit compression).

As we have assumed a 128-bit SIMD execution model, the concatenation has to done by hardware using one 128-bit SIMD register-concatenate instruction. Figure 5 illustrates this operation.

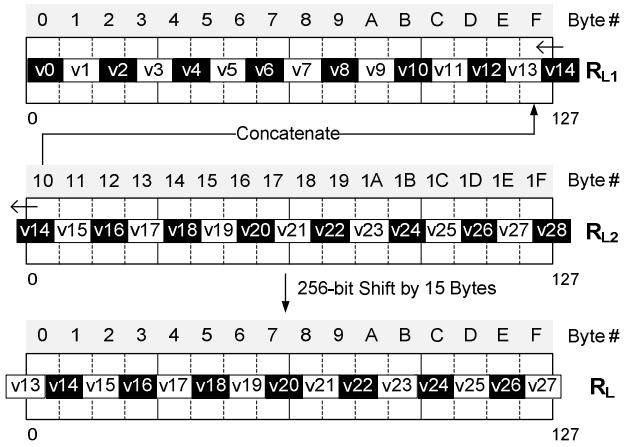

Figure 5. Concatenating two SIMD registers for value alignment in one hardware instruction

4.1.2 4-Byte Alignment

After having a 128-bit block of data in the SIMD register R_L, four compressed integer values are copied to four separate 4-byte DWs in a new register R_C. This is needed because four decompressed values need four 32-bit storage spaces. Also, this provides an efficient way to decompress the four values into their corresponding DWs and finally store them back as a single batch.

Typically, the SIMD instruction-set offers a selective copy (shuffle) instruction that allows copying a group of specific bytes (or words) from the source register to a specific bytes (or words) in the target register. This mapping is done using a copy $mask$. Figure 6 shows the copying step of the loaded data using a single 128-bit SIMD shuffle instruction.

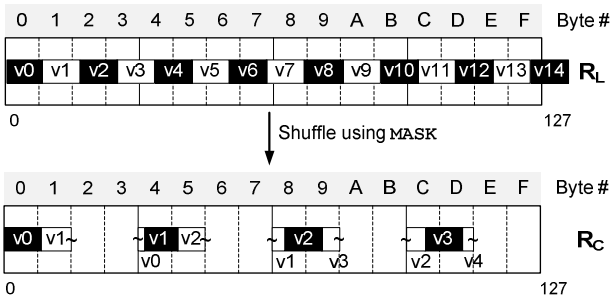

Figure 6. Copying four 9-bit values to separate DWs

In Figure 6, MASK ($source\_byte$, $dest\_byte$) = {(0,0), (1,1), (1,4), (2,5), (2,8), (3,9), (3,C), (4,D), ($zeros$, $else$)}. Additionally, only the values in R_C that are marked in black are valid**.** A careful inspection in the figure shows that not all values are aligned at the beginning of their corresponding DWs (unlike other cases; for example if 8-bit compression is used). Also, there are invalid bits trailing the actual values that have to be masked out (i.e. set to zero). These issues are handled by the next decompression step (the Bit-Alignment by value extraction and storing).

Another issue that might arise is that a single value might span across DWs, that is, the value is packed in more than 4 bytes in R_L. Figure 7 gives an example for value-spanning issue with 27-bit compression.

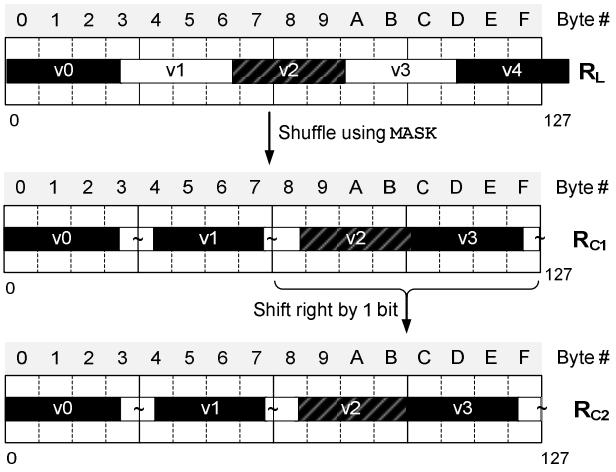

Figure 7. Spanning-value issue with 27-bit compression

In this figure, the third value in R_L occupies the 7th byte till the 11th byte (with the first 6 bits and the last 7 bits being invalid) while spanning across 5 bytes. This is an issue because there is no way to copy 5 bytes into one DW. One solution is to actually copy the values to separate DWs (while one of them spanning across its DW to the next) and then shift and combine the DWs in a way that each value is in exactly one DW. However, additional SIMD logic instructions are required. This is achieved in Figure 7 by using a MASK ($source\_byte$, $dest\_byte$) = {(0,0), (1,1), (2,2), (3,3), (3,4), (4,5), (5,6), (6,7), (6,8), (7,9), (8,A), (9,B), (A,C), (B,D), (C,E), (D,F)} and then shifting the upper QW to the right by one bit. As a result, additional overhead is expected to solve this issue that could cause non-linear speedup for the decompression routines (which is shown in the evaluation results).

4.1.3 Bit Alignment

The goal of this step is to align each one of the four values in R_C at the first bit of their corresponding DWs and mask out the trailing invalid bits, thus decompressing them into four equivalent 32-bit integers in the result register R_E. This is also needed to have a direct way to store the values back in the memory in a single SIMD store instruction.

In order to do so, a 32-bit SIMD shift instruction, with four variable shift amounts, is needed to align the values at the same position in their corresponding DWs. After that, a 128-bit SIMD AND instruction with an appropriate mask operand is needed to finish the extraction by masking out the invalid bits (i.e. by setting them to zero). As a result, ideally two 128-bit SIMD instructions are needed. Figure 8 visualizes the extraction process. In this figure, all values are independently left-shifted so that they are all positioned at the beginning of their DWs. Finally, all invalid bits are cleared out.

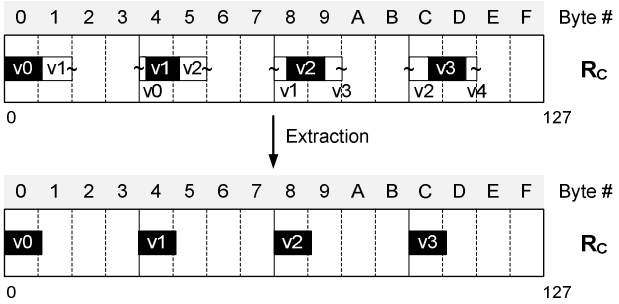

Figure 8. Extracting the four 9-bit values

As a result of the value extraction by aligning the values at the bit-level, a direct 128-bit SIMD store instruction is needed to store back the four decompressed 32-bit integers in a temporary memory location. These decompressed integers can now be used by the query execution engine to execute or continue its operation. Figure 9 depicts the storage step.

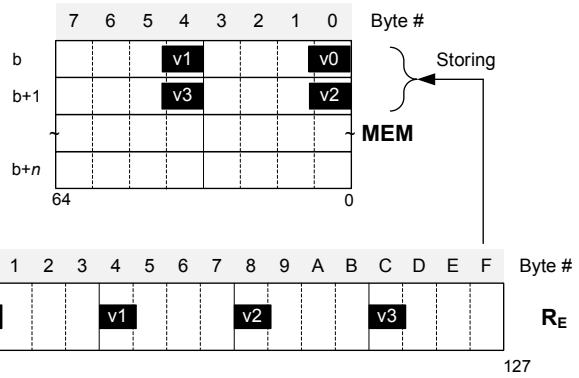

Figure 9. Storing the decompressed integer values

4.2 Vectorized Predicate Handling

Often, the query processing engine determines that only a specific value or value-range is needed to process a query as a part of the table scan operation. Normally, a search algorithm returns the indexes of the compressed values that fall in the search range. Alternatively, a bit-vector of the compressed values can be generated that maps the search result in a bitwise manner. For example, if the compressed data represents an array of ascending integers (indexes) starting from "0_d" till "255_d" and the search condition is the integer range of "1_d" to "5_d", the resulting 256 bit-vector would be "$011111000\ldots0_b$" and the index array would be "1_d, 2_d, 3_d, 4_d, 5_d" for a 0-indexed array.

Rather than decompressing the whole data then start searching for the values, direct LWC-NC compressed comparison with a search condition (i.e. predicate) can be used instead. Here, bit alignment is not needed as the unaligned values (but still 4-byte aligned) could be directly compared to the search range through shifting the search values by the same amount. Figure 10 illustrates how our SIMD approach can be adapted to search for a range of values (min, max) in LWC-NC compressed columns.

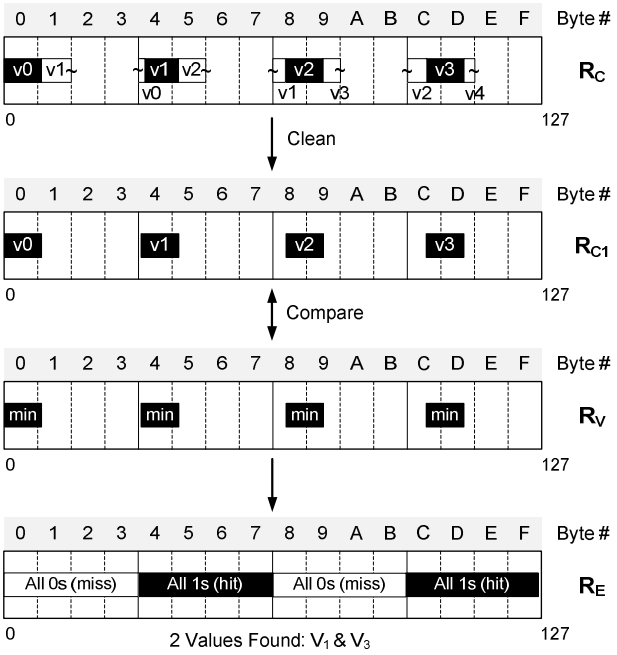

Figure 10. Vectorized scan-value search with predicate

In this figure, the four 4-byte aligned values in R_C are compared to the min value in a single 32-bit SIMD compare instruction. The same process is executed for the max value and is combined with the previous comparison result (by a single 128-bit SIMD AND instruction). Thus, the result register will hold four search results - each held in its corresponding DW. The DW is set to zero if the value was not found (i.e. v_i didn't fall in the range). This can be used to generate an index array or a bit-vector. It is worth mentioning that converting the scan result into a bit-vector can be vectorized. However, this is beyond the scope of this paper.

The general *SIMD-Search Algorithm* for a search interval of $[min, max]$ works as follows:

parallel_shift (min, min, min, min) **by** $(s_{a0}, \ldots, s_{a3})$, **store** in min_a
parallel_shift (min, min, min, min) **by** $(s_{b0}, \ldots, s_{b3})$, **store** in min_b
parallel_shift (max, max, max, max) **by** $(s_{a0}, \ldots, s_{a3})$, **store** in max_a
parallel_shift (max, max, max, max) **by** $(s_{b0}, \ldots, s_{b3})$, **store** in max_b
set k **to** 0
for i **from** 0 **to** $max\_index/128$ {
  for j **from** 0 **to** 15 {
    parallel_load b_a **from** $input[k * 16 + j * n]$
    shuffle b_a **to** c_a **using** $shuffle\_mask(m_{a0}, \ldots, m_{a15})$
    parallel_compare c_a **with** (min_a, max_a), **store** in t_a
    convert t_a **to** 4-bit integer r_a
    parallel_load b_b **from** $input[k * 16 + j * n + n/2]$
    shuffle b_b **to** c_b **using** $shuffle\_mask(m_{b0}, \ldots, m_{b15})$
    parallel_compare c_b **with** (min_b, max_b), **store** in t_b
    convert t_b **to** 4-bit integer r_b
    store $(r_a \ll 4) + r_b$ **in** $output[i * 16 + j]$
  }
  increase k **by** n
}

Where n denotes the maximum number of bits as computed by Equation 1. The variables min_a, min_b, max_a, max_b, b_a, c_a, b_b, and c_b are vector variables holding four integers. Note that the scan results, which could be transformed into a more efficient form (like a bit-vector), are stored in the $output$ array.

5. IMPLEMENTATION

Many implementations of SIMD exist in the market like Intel® SSE [14] and AMD® 3DNow! [15]. We chose Intel's SSE implementation (SSSE3 and SSE4.1) as it provides most of the needed SIMD features that we have discussed in Section 2 and Section 4. In this implementation, 128-bit SIMD registers are used and aligned memory access is assumed. We used the SIMD intrinsic programming method for an optimized implementation by following the guidelines discussed in [16]. Also, as SSE uses 128-bit execution environment, the implementation is a direct reflection of the discussed concepts in Section 4.

The 16-byte alignment (i.e. loading) step discussed in Section 4.1.1 is implemented by a single 128-bit SIMD load instruction. To optimize the execution, we used memory pre-fetching to fetch a set of compressed 128-bit block in advance, thus, further improving the local cache performance. Unaligned memory access is avoided in architectures that force access penalties (i.e. pipeline stalls) by using a 128-bit SIMD concatenate/shift instruction that implements the solution provided in Section 4.1.1. This SIMD instruction performs both concatenation and shifting in the same hardware instruction and produces a 128-bit result. It should be noted that new Intel architectures such as the one used in Intel® Xeon® Processor 5500 series, provide fast unaligned SIMD data access (same as aligned access) which is practically preferred as it simplifies the next decompression step.

4-byte alignment by selectively copying the values to separate DWs as discussed in Section 4.1.2 is implemented by a single 8/16-bit SIMD shuffle instruction with an appropriate shuffle mask. The spanning-value solution is also implemented using the same concept discussed in Section 4.1.2, by using one 8-bit SIMD shuffle instruction and four 32/64/128-bit SIMD logic instructions. Another optimized implementation uses 8/16-bit SIMD shift instructions and 128-bit SIMD blend instructions to achieve the same solution in much less latency.

For the bit alignment (i.e. extracting and storing) step discussed in Section 4.1.3, we realized the independent 32-bit SIMD shift operation (i.e. four different shift amounts for each DW) by using 32-bit SIMD integer multiplication. The idea is that multiplying a value by "2_d" results in 1-bit shift to the left. So to shift left an operand by n-bit, the second operand (multiplicand) has to be 2^n. After shifting all values so that they have the same number of preceding (invalid) bits m, a single 32-bit SIMD shift instruction is used to shift all DWs by m so that all values are align at the beginning. Hence, this implementation realizes an independent shift (to the right) by only two SIMD instructions. After that, the invalid bits are masked out by using a single 128-bit SIMD AND instruction with an appropriate mask operand. Finally, the decompressed values are stored back in the memory by using a single 128-bit SIMD store instruction.

It should be noted that our implementation has many other alternatives for bit alignment. Each alternative has its own speedup advantage but is applicable to specific compression-bit case. Integer multiplication is the mostly used implementation and delivers near-best speedup. If only two distinct shift amounts are required, like in 4-bit and 6-bit compression cases, it is beneficial to use a shift instruction and a blend instruction to realize the same concept. With this technique, more care is needed in the proceeding 8/16-bit SIMD shuffle instruction to arrange the values in the correct order. It is also worth noting that a 128-bit SIMD compare instruction with a bit mask can be used for 1-bit compression to spread the value of a bit to all bits in the same byte (i.e. extend the bit to the byte level). Hence, a single comparison can therefore be used to expand the values. We have also evaluated other implementations using division, addition, shuffle, and logic SIMD instructions to realize the same concept and work on 2, 4, 8, and even 16 compressed values at a time. However, these implementations are slower and we list them only for the sake of completeness. Also, not all SIMD architectures support all of the assumed instructions so that they need to be implemented differently. The alternatives that turned out to be the fastest are the ones that we have described in detail here and in Section 2.

For table scan search with predicated discussed in Section 4.2, we used two 8/16/32-bit SIMD integer compare instructions to build up the search result. One additional 128-bit SIMD AND instruction is used to format the search result as all 1s or 0s to simplify the index or bit-vector generation. Using the 128-bit SIMD move/mask instruction, the SIMD result can be converted efficiently to a scalar mask m, which can be written to a bitvector. Alternatively, the scalar result can be used to generate the indexes of the search results. This can be implemented efficiently by maintaining a SIMD register with the current indexes and using 8/16/32-bit SIMD shuffle instructions for storing the result. In this case, the scalar mask m can be used as an index for a look-up table holding possible shuffle masks.

We integrated our approach into SAP® Netweaver® Business Warehouse Accelerator (BWA) [17]. SAP® Netweaver® BWA is an appliance-like solution co-developed by SAP and Intel. The software indexes selected information to create a highly compressed index structure that loads to the memory whenever users request the data. The accelerator uses high-performance aggregation techniques to process queries entirely in memory, and then delivers results back to the SAP® Netweaver® BW for output to users.

6. EXPERIMENTS

For our evaluation, the SAP® Netweaver® BWA engine was modified in a way that either the standard or the vectorized table scan method can be used. In order to present realistic results, we did deep (production like) integration on the engine level without the need for further data copies or data transformations during query runtime. Furthermore, the engine implements two versions of the full table scan; the first method only decompresses the table column, while the second method integrates search predicate handling into the scan without unpacking the data in advance. For both versions we implemented a SSE version based on our approach.

We implemented the evaluation experiments on a single server equipped with two Intel® Xeon® Processors X5560 (2.8GHz), each having four processing cores and 8MB last level cache. The server was equipped with 24GB of RAM and the operating system was SuSE* Linux* Enterprise Edition 10, Service Pack 2.

For each compression-bit case (determined by Equation 1), 1B-records were decompressed 10 times for each implementation and the running time was recorded. The performance of the decompression routine is mostly data-independent and varies only with the used compression-bit case.

Figure 11 depicts the median query time for each bit case using different implementations of the decompression routine. There, it clearly seen that the existing table scan method is already optimized for performance by minimizing the cache miss rate and massively unrolling the code loops, which allows the pre-computation of shift arguments and masks. As a reference point, we also included the results for a variant without loop-unrolling that shows a significantly higher latency.

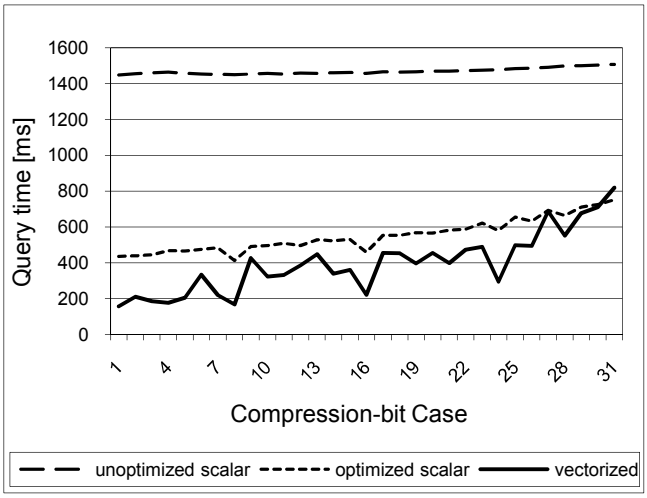

Figure 11: Time to decompress 1B integers

Also in Figure 11, it should be noted that the performance gain is significant as we test against a fully optimized system implementation (over years). In this implementation, the high performance of the full table scan is considered to be one of the main values of the system. We achieve performance improvements on top of that using our prototype implementation, which demonstrates that our approach is indeed promising.

The speedup of the SIMD implementation for the value decompression, against the highly optimized scalar version, is

shown in Figure 12. The performance improvement is generally higher for the bit cases up to 8 bits, where 8 values can be processed in parallel in one SSE register. There, the average speedup factor is 1.58 over all bit cases.

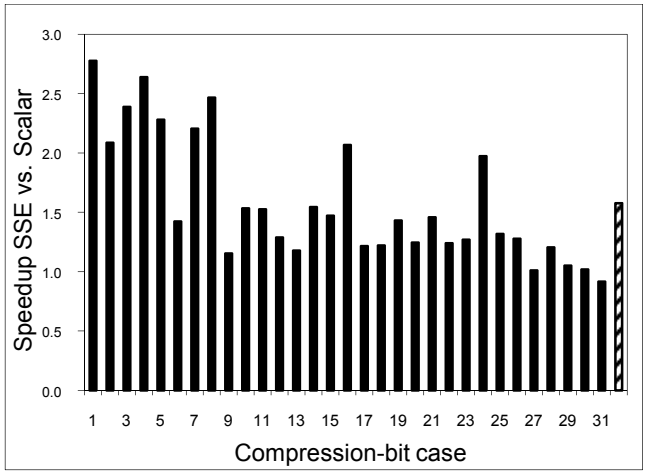

Figure 12. Speedup for decompression by vectorization

The speedup of the SIMD implementation for searching a value (full-table scan) in 1B records is shown in Figure 13. The experimental test-set for bit case n consists of the natural numbers modulo 2^{n+1}. Again, the measurements were performed 10 times on a test program executing the search routine as described in Section 4.2, and the median of the 10 runs was used for computing the speedup. For the lower bit-compression cases, the search result is very large for a single search-value (e.g. if 2 bits are used, a quarter of our test data set is returned). For bit cases 27 onwards, special care is needed to handle compressed values that span across 5 Bytes as shown in Figure 7. As a result, this reduces the performance advantage to the extent that for bit case 31, the vectorized implementation was slower than the scalar version. However, the average speedup factor of a full-table scan is still 2.16. In practice, the SIMD implementation is only used in bit cases where it is faster than its scalar counterpart, which is the dominant scenario.

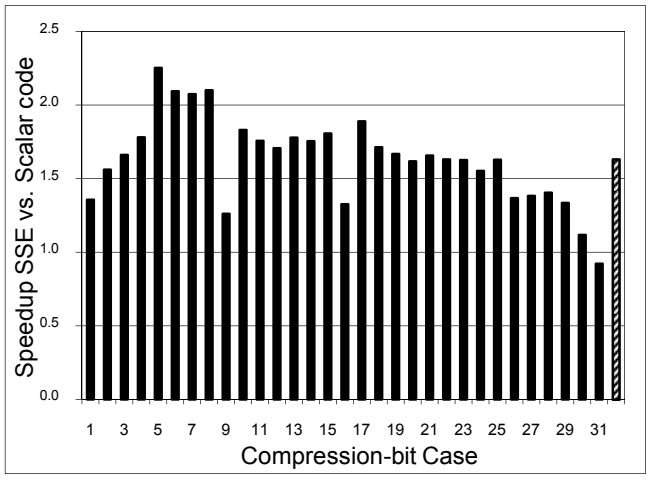

Figure 13. Speedup of full-table scan by vectorization

If the result of a full table scan is returned as a bit-vector, the running time is independent of the number of hits. However in case a list of indexes is returned, the running time increases for large results as storing the results cannot fully exploit the benefit of storing vector instructions. The best speedup is therefore achieved for very selective queries as graphed in Figure 14, which displays the Speedup vs. Selectivity. Again 1B entries were processed 10 times and the median was recorded. Each point in the graph displays the average speed-up over all bit cases. The overall speedup average is 1.63.

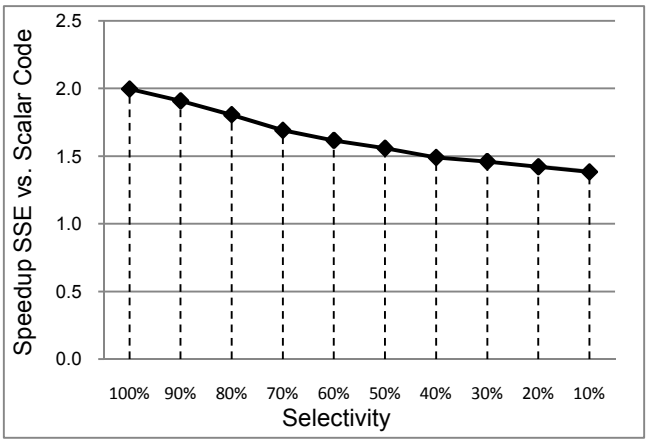

Figure 14. Speedup of full-table scan by selectivity

In real world scenarios, and according to our experience at SAP, the compression bits used to compact database columns are mainly in the range of 8 to 16 bits. Figure 15 shows the practical distribution of the compression bit cases against the running time contribution of the table scan routines for a typical customer scenario. Taking this distribution into account, the (weighted) speedup factor for a full-table scan is 2.45 over all bit cases.

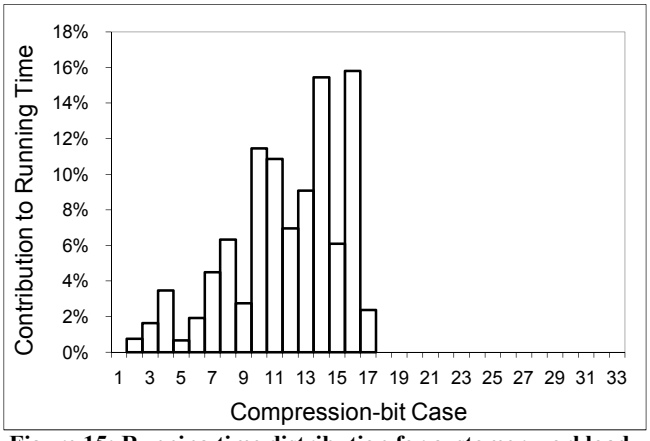

Figure 15: Running time distribution for customer workload

Finally, we executed the vectorized search in parallel on different processor cores to verify its scalability. Figure 16 shows that the vectorized search scales almost linearly up to eight cores that are installed on the evaluation system. The memory bandwidth leaves sufficient headroom for future processors with

more than four cores per socket. It should be noted that compressed data is very SIMD-unfriendly because it is completely unaligned and data elements do not even begin at byte boundaries. The core result of the paper is that even in this case, SIMD can result in significant speedups.

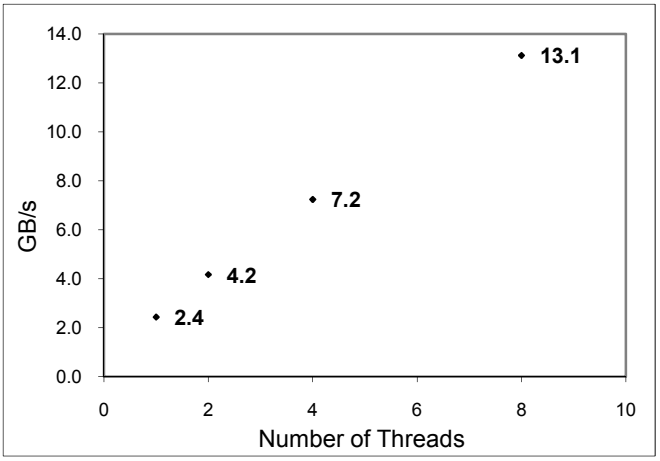

Figure 16. Scalability of vectorized full table scan

It is worth mentioning that we did an evaluation using graphics card (a GPU port) and other vector processing enabled hardware accelerators. GPUs or other specialized hardware are currently limited by the PCIe throughput of about 4 GB/s and the typical customer workload which is too big to be stored on the graphics card local memory. As shown in Figure 16, we are processing 13.1 GB/s (when executed in parallel), which therefore cannot be achieved on a GPU when the data transfer is taken into consideration. In contrast, we focus in this paper on demonstrating the concepts and principles needed to achieve a vectorized table scan using off-the- shelf CPUs.

7. CONCLUSION

Main memory column-store database systems rely on full table scan to avoid expensive indexing, and hence, reduce memory consumption. These operations operate on highly compressed data and are CPU-bound like never before.

In this paper, we proposed a SIMD approach to accelerate main memory table scan operations using on-chip vector processing units. Our solution targets database systems working on highly compressed in-memory columns and does not require any architectural changes. In the evaluation section, we proved that our solution efficiently vectorize the decompression and scan-value search operations (full table scan) with high granularity. We showed that our approach considerably accelerate table scans and scales well with the number of cores. Consequently, it adds to and complements the already existing SMP optimization of table scan operations.

Acknowledgement: We would like to thank Franz Färber, Günter Radestock, Tobias Mindnich, and Christoph Weyerhäuser from SAP AG for the fruitful discussion and the tremendous help in integrating and testing the SIMD routines.

8. REFERENCES

[1] Westmann, T., Kossmann D., Helmer, S., Moerkkotte, G., "The Implementation and Performance of Compressed Databases," in *SIGMOD*, vol. 29, no. 3, pp. 55-67, 2000

[2] Harizopoulos S., Liang V., Abadi D., Madden S., "Performance tradeoffs in read-optimized databases," In *VLDB*, pp. 487–498, 2006

[3] Flynn, M.J., "Very high-speed computing systems," Proceedings of the IEEE , vol.54, no.12, pp. 1901-1909, 1966

[4] Duncan, R., "A survey of parallel computer architectures," Computer , vol.23, no.2, pp.5-16, Feb 1990

[5] Graefe, G., Shapiro, L.D., "Data Compression and Database Performance," Applied Computing, pp. 22-27, 1991

[6] Zukowski M., Heman S., Nes N., Boncz P., "Super-Scalar RAM-CPU Cache Compression," *Data Engineering, International Conference*, vol. 0, no. 0, pp. 59, 2006.

[7] Holloway A., Raman V., Swart G., DeWitt D., "How to Barter Bits for Chronons: Compression and Bandwidth Trade Offs for Database Scans," In *SIGMOD*, pp. 389-400, 2007

[8] Qiao, L., Raman, V., Reiss, F., Haas, P. J., and Lohman, G. M., "Main-memory scan sharing for multi-core CPUs," In *VLDB*, pp. 610-621, 2008

[9] Johnson, R., Raman, V., Sidle, R., and Swart, G., "Row-wise parallel predicate evaluation," In *VLDB*, pp. 622-634, 2008

[10] Zhou J., Ross K.A., "Implementing database operations using SIMD instructions," In *SIGMOD*, 2002.

[11] Heman S., Nes N., Zukowski M., Boncz P., "Vectorized Data Processing on the Cell Broadband Engine," Data Management on New Hardware, no. 4, 2007

[12] Roth M., Van Horn S., "Database compression," In *SIGMOD* Record, pp. 31-39, 1993

[13] Goldstein J., Ramakrishnan R., Shaft U., "Compressing relations and indexes," In *ICDE*, 1998

[14] Abel J., Balasubramanian, K., Bargeron M., Craver T., Phlipot M., "Applications Tuning for Streaming SIMD Extensions," Intel Technology Journal Q2, 1999

[15] Oberman S., Favor G., Weber F., "AMD 3DNow! Technology: Architecture and Implementations," IEEE Micro, vol. 19, pp. 37-48, 1999

[16] Gerber R., Bik A., Smith K., Tian X., "The Software Optimization Cookbook," 2nd edition, Intel Press

[17] SAP AG, https://www.sdn.sap.com/irj/sdn/bia

[1] Intel and Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States or other countries. Other names and brands may be claimed as the property of others.

[2] SAP and SAP Netweaver are registered trademarks of SAP AG or its subsidiaries in Germany and several other countries. Other names and brands may be claimed as the property of others.